Designing a high-speed asynchronous 80C51 microcontroller

van Hoek, T.J.H.

Award date:
2010

TU/e technische universiteit eindhoven

Faculty of Electrical Engineering
Section Electronic Systems (ICS/ES)
ICS-ES 902

Master's Thesis

DESIGNING A HIGH-SPEED ASYNCHRONOUS 80C51 MICROCONTROLLER.

T.J.H. van Hoek

Supervisor:	prof.dr.ir. J. Pineda de Gyvez/prof.dr. K. Van Berkel
Coach:	A. Brink (Handshake Solutions)
Date:	December 2008

Document Information		
Document title	Designing a high-speed asynchronous 80C51 microcontroller – Master's Thesis TU/e	
Date of creation	04/12/2007	
Date of last change	09/12/2008	
File name	HT80C51-HS	
Status	Release	
Version number	1.0	
Steering group	Project Owner : Project Manager : Project Mentor : Project Supervisors :	Rik van de Wiel (HS) Ad Peeters (HS) Arjan Bink (HS) Kees van Berkel & Jose Pineda de Gyvez (TU/e)
Client / Target audience	Eindhoven University of technology Faculty of Electrical Engineering Section of Embedded Systems	
Summary	This master's thesis presents and explains the performance improvements made to an asynchronous HT80C51 microcontroller core and limitations found in the Handshake Technology design flow. First the synchronous 80C51 and the initial design of the asynchronous HT80C51 microcontroller will be described and analyzed. The bottlenecks in the design and architecture of the microcontroller will be identified and documented. Second, this Master's Thesis introduces a new asynchronous architecture for the HT80C51. The new design will be explained, verified, benchmarked and documented.	
Keywords	Handshake Technology, asynchronous, high-speed, 80C51, microcontroller	
Contact	**Handshake Solutions** High Tech Campus 12 5656 AE Eindhoven The Netherlands	phone: +31-40-27 46114 fax: +31-40-27 46526 info@handshakesolutions.com www.handshakesolutions.com

Foreword

This master's thesis describes the speed improvements made to an asynchronous 80C51 microcontroller core, known as the HT80C51 (Handshake Technology 80C51).

The HT80C51 is the Handshake Solutions (refer to A1) implementation of the 80C51 8-bit microcontroller. As with all Handshake Technology implementations, the HT80C51 boasts ultra low operational power consumption, zero active stand-by power and immediate wake-up. Furthermore, very low electromagnetic emission (EME) levels and current peaks enable easy integration with RF and analog circuitry.

The HT80C51 delivers unique benefits to any application where a clocked 80C51 core can be used, particularly when power consumption or electromagnetic interference is an important issue. Handshake Technology implementations of the 80C51 have been used in numerous ICs across various markets including wireless, smart cards and automotive.

The motivation for this thesis is that current asynchronous designs using Handshake Technology (e.g. ARM996HS, HT80C51) are outstanding regarding power and EME, but lag in terms of speed performance. These asynchronous designs are based on Handshake Solutions' design environment (refer to chapter 2).

Handshake Solutions provided me the possibility to start a part-time study at Eindhoven University of Technology. I enjoyed the courses I had at the University. It helped me to develop myself as an engineer and find a wider and deeper understanding of the fundamentals of computer engineering, the theory of information and communication systems and the complexity of electronic designs.
The study at Eindhoven University of Technology helped me to solve problems by first dividing them into clear defined cases and then solving them by abstract and structural methods.

It is my hope that this thesis will be useful to Handshake Solutions and Eindhoven University of Technology.

Table of Contents

Abstract

This master's thesis describes the speed improvements made to an asynchronous 80C51 microcontroller core known as HT80C51. The motivation for this assignment is that current asynchronous designs using Handshake Technology (e.g. ARM996HS, HT80C51) are outstanding with respect to power and electromagnetic emission (EME), but lag in terms of speed performance.

The assignment is to make a high speed version of the HT80C51. With the knowledge of the HT80C51 in the Handshake Solutions group, the focus is to explore the architecture of the HT80C51 by asynchronous pipelining while staying completely in the Handshake Technology design flow.

First the current 80C51 implementations (both synchronous and asynchronous) are analyzed. The available HT80C51 microcontroller is benchmarked and the architecture is analyzed. Bottlenecks of the HT80C51 microcontroller are identified and solutions are found to reach the goal of obtaining higher performance.

A high-speed asynchronous 80C51 microcontroller is developed and analyzed in the Handshake Technology design flow. The design is implemented and validated in a CMOS 0.14 µm technology library. The performance of the high-speed asynchronous 80C51 is 3.2 times faster than the current asynchronous HT80C51 design and up to par with current high-speed synchronous designs. This makes it more than plausible that it is possible to not only benefit from typical characteristics of asynchronous technology like low power, but that it is also possible to make fast digital implementations with asynchronous technology.

Acknowledgements

It is arguable that this page is the most widely read page from the whole thesis. I would like to think that is not the case in this thesis, but who I am fooling?

I would like to start thanking my colleague at Handshake Solutions, roommate, friend and mentor in this thesis, Arjan Bink. If I could not benefit from your experience, knowledge and time, I always could enjoy your wide and good taste of music.
This certainly also applies for Ad Peeters. I am admiring your patience and willingness to guide me and this thesis and I am very thankful with all your suggestions and support.
Thank you to Kees van Berkel and Jose Pineda de Gyvez for your help and time for supervising this thesis work.
Thanks to Rik van de Wiel who always supported my part-time study.
I will not leave out all my colleagues at Handshake Solutions; Bert, Erwin, Frank, Frits, Jur, Klaus, Marc, Marc, Maria, Mark, Marten, Monique, Nidish, Paul, Rachna, Rene, Rob and Wouter.

Finally yet importantly, thanks to my parents who always gave me the warm and proud feeling good parents give. No matter what I accomplished, they always supported me and were proud of me. The same holds for my whole family and grandmother.
Thanks to all my friends who always supported me with laughs, distraction, sports and other ways to expand the good life I am living.
Finally I want to thank, two very important persons in my life, more personally. My brother Berry, who is more than just a Big Brother for me! Not only a friend or just an older brother, but someone I would never want to miss in my life. Thanks to my lovely Miriam for withstanding with me in the hectic finale of my study. I can not wait to spend more time with you and enjoy your good presence. Thanks that you are expanding my life!

Abbreviations and Terminology

ADD	ADDITION instruction, this instruction adds two bytes and write the result to the destination
ALU	Arithmetic and Logic Unit
Argument	Instruction extension (address or constant)
ASIC	Application Specific Integrated Circuit
Branch	A Branch is point in a computer program where the flow of control is altered
Code Memory	Memory space that is accessed for fetching instruction opcodes and their arguments
CPU	Central Processing Unit
Data Memory	Memory space that is accessed for reading and writing SFRs and temporary data
DPTR	Data Pointer
EB	Execution Bits (Condition Bits)
EDA	Electronic Design Automation
EME	Electromagnetic Emission
Haste	Handshake Solutions programming language for designing asynchronous circuits
Htcomp	Handshake Technology COMPiler
Htmap	Handshake Technology MAPper
Htprof	Handshake Technology PROFiler
Kbyte	Kilo Byte – 1024 Bytes
MOV	MOVE instruction, this instruction moves one byte from the source to the destination
MIPS	Million Instructions Per Second
Opcode	Instruction identifier (Operation Code)
PC	Program Counter
PSW	Program Status Word
RAM	Random Access Memory
ROM	Read Only Memory
SFR	Special Function Register
SP	Stack Pointer
TiDE	Timeless Design Environment, Handshake Technology design flow to design asynchronous circuits
VHDL	VHSIC Hardware Description Language
VLSI	Very Large Scale Integration.

1. Introduction

This master's thesis presents the results obtained from research at the company Handshake Solutions. I have carried out the work for this project from December 2007 until December 2008. Kees van Berkel and Jose Pineda de Gyves were the supervisors at Eindhoven University of Technology and Ad Peeters and Arjan Bink were the supervisors at Handshake Solutions.

1.1. Problem description

Handshake Solutions has developed the HT80C51: an 80C51 8-bit CISC microcontroller based on Handshake Technology that is functionally compatible with the original Intel 8051.
Current designs of Handshake Solutions (e.g. ARM996HS, HT80C51) have been developed with excellent power consumption and EME characteristics, but lag in terms of performance.

The target of this graduation project is:

- Analyze and benchmark the current HT80C51 implementation
- Propose a new architecture which can reach an instruction throughput that is at least a factor 3 higher compared to the current HT80C51. The new architecture is fully designed in Handshake Solutions' design flow, and should be identically compiled as the current HT80C51
- Implement the new HT80C51 architecture
- Measure and analyze the new HT80C51 architecture
- Comment on the new architecture
- Document the approach to the new design

The current HT80C51 architecture is designed with area and power consumption as main design goals. Performance never was the main design goal. The choice to use the HT80C51 for achieving performance is justified for this type of microcontroller.

1.2. Problem approach

To design a high-speed HT80C51 microcontroller a high-level programming language, called *Haste*, is used. Haste is part of the Handshake Technology design flow. With this design flow it is possible to generate a Handshake circuit and a netlist for a given standard-cell library (like TSMC, etc).

According to the problem description we aim at optimal results in terms for speed performance. Naturally the microcontroller should be functionally correct, which means that it executes instructions without any errors. Low area and power consumption are less important.

First the standard synchronous 80C51 and the current asynchronous HT80C51 implementation are analyzed. With knowledge of the synchronous 80C51 instruction set and memory management we determine the maximum possible performance without changing these two main parameters. With the HT80C51 analysis we evaluate design bottlenecks and find another design approach more suitable for high-speed operation.
With the knowledge gathered by the previous design analysis we determine a better performing architecture for a high-speed HT80C51.
After implementing the high-speed HT80C51 architecture steps are needed to improve, tweak and balance this improved architecture.

1.3. Document overview

This master's thesis describes the functional design and implementation of a high-speed asynchronous 80C51 compatible microcontroller core.

Chapter 1 defines the problem description, our main target and problem approach. This chapter also gives an overview of this thesis.

Chapter 2 is an introduction to Handshake Technology (HT). This chapter introduces an overview of the Handshake Technology design flow *TiDE*, Handshake Technology programming language *Haste* and handshake circuits.

Chapter 3 presents the 80C51 microcontroller in five steps. First of all, the original Intel architecture is introduced. Second, other synchronous architectures of the 80C51 are discussed. Third, the first asynchronous implementation by Hans van Gageldonk [2] is presented. Fourth, based on an analysis of this design, a new design has been created that is known as the HT80C51-LC (Low Cost). The HT80C51-LC design will be discussed and serves as the starting point for this thesis work. Fifth, the last section of this chapter will be dedicated to the theoretical maximum performance that can be achieved for an 80C51.

Chapter 4 describes an initial asynchronous speed-up version, the HT80C51-SU (Speed Up). This initial design of the high-speed HT80C51 implements a pre-fetching unit, a multi stage decoder and an optimized execution phase. The main goal was to increase the performance without a major design change. This was done by reducing the overhead of control logic in the execute phase.

Chapter 5 introduces a complete new architecture for an asynchronous 80C51 design, the HT80C51-HS (High Speed). This design introduces a pipelined architecture of the asynchronous 80C51. This design choice was made to increase the maximum performance with a vast amount. Other techniques are implemented to boost the performance of the HT80C51 to the desired level. These techniques are discussed and evaluated in this chapter. This chapter also outlines some problem areas in the Handshake Technology design flow. Some of these problems should be addressed to increase speed performance

Chapter 6 outlines the results obtained from this graduation project. Conclusions are drawn and the results are qualified. Chapter 6 finalize with a summarization about the work activities that still needs to be done and recommendations to further improve the high-speed asynchronous 80C51 microcontroller in the future.

2. Handshake Technology

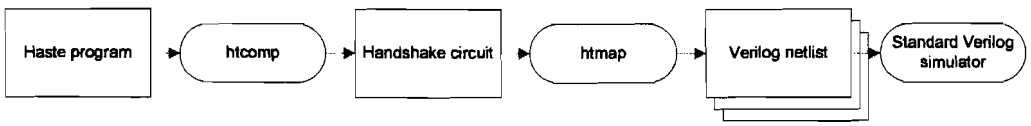
2.1. Handshake Technology design flow

The Handshake Technology design flow (TiDE) targets at efficiently bringing a design from specification to a self-timed implementation. This has led to a development of a complete design flow consisting of compilation, simulation and analysis tools which supports the designer in the design process. This design flow goes all the way from a high-level design entry down to a circuit-level realization. The design flow has been set up in such a way that new tools are developed only where needed. Otherwise, standard Electronic Design Automation (EDA) tools are being used. Additional tools make it possible to bring the initial design to a final layout.
The Handshake Technology design flow can be seen as an alternative front-end to the standard EDA flows, complementary and compatible with them.

[Figure 1] Design flow principle

2.2. Functional design flow

The functional part of the design flow, during which the designer's focus is on the functional correctness of the design, is shown schematically in [Figure 2]. In this diagram, boxes denote design representations and test benches, while the oval boxes refer to tools.
The two central tools are htcomp, which compiles a Haste program into a handshake circuit and htmap, which compiles a handshake circuit into a Verilog netlist.



[Figure 2] Functional design flow overview

2.2.1. Design language Haste

Ever since the introduction of the relatively new asynchronous VLSI programming paradigm throughout the world, people consider designing on the level of asynchronous circuits inherently difficult. To abstract from the properties of asynchronous design (e.g. intricate timing behaviour, difficult to control delays, etc.), Handshake Solutions has designed a dedicated programming language called *Haste*, formerly known as Tangram. Haste is more a behavioural design language than a structural design language. The programmer has the opportunity to express the most complex algorithms using powerful constructs (e.g. parallelism and channel communication). Haste has similarities with C, and is comparable to behavioural level Verilog or VHDL. A complete overview of the design language Haste can be found in [4]. Below we highlight some key aspects that make Haste a unique and powerful design language.

[Code Fragment 1] shows the sequential composition command. The *semicolon* symbol represents that command C_0 must execute (and finish) before command C_1 executes.

```
C0; C1
```

[Code Fragment 1] Sequential composition command

[Code Fragment 2] shows the parallel composition command. The *two vertical lines* symbol represents that both commands C_0 and C_1 execute simultaneously.

```
C0 || C1
```

[Code Fragment 2] Parallel composition command

[Code Fragment 3] and [Code Fragment 4] shows the selection command. The Boolean expressions B_0 through B_n are guards. All guards evaluate simultaneously and the selection command selects the command to be executed corresponding to the first guard in the list that evaluates to true. There are two possible selection criteria. First possibility [Code Fragment 3] is that if all guards are false, it selects then none of the commands and finishes immediately.

```
if   B0 then C0
or   B1 then C1
...
or   Bn then Cn
fi
```

[Code Fragment 3] Selection command

Second possibility [Code Fragment 4] is that if all guards are false, the command blocks until at least one of the guards evaluates to true and therefore always selecting one of the commands (C_0 through C_n).

```
sel  B0 then C0
or   B1 then C1
...
or   Bn then Cn
les
```

[Code Fragment 4] Selection command with wait condition

[Code Fragment 5] shows the infinite repetition command, which executes command C, and continuously restarts execution of it as soon as C finishes.

```
forever do
  C
od
```

[Code Fragment 5] Infinite repetition command

[Code Fragment 6] shows the assignment command. The number of bits for variables V_0 through V_n must be equal to the number of bits for expressions E_0 through E_m. Also the types of variables V_0 through V_n must be equal to the types of expressions E_0 through E_m.

```
<< V0, ..., Vn >> := << E0, ..., Em >>
```

[Code Fragment 6] Assignment command

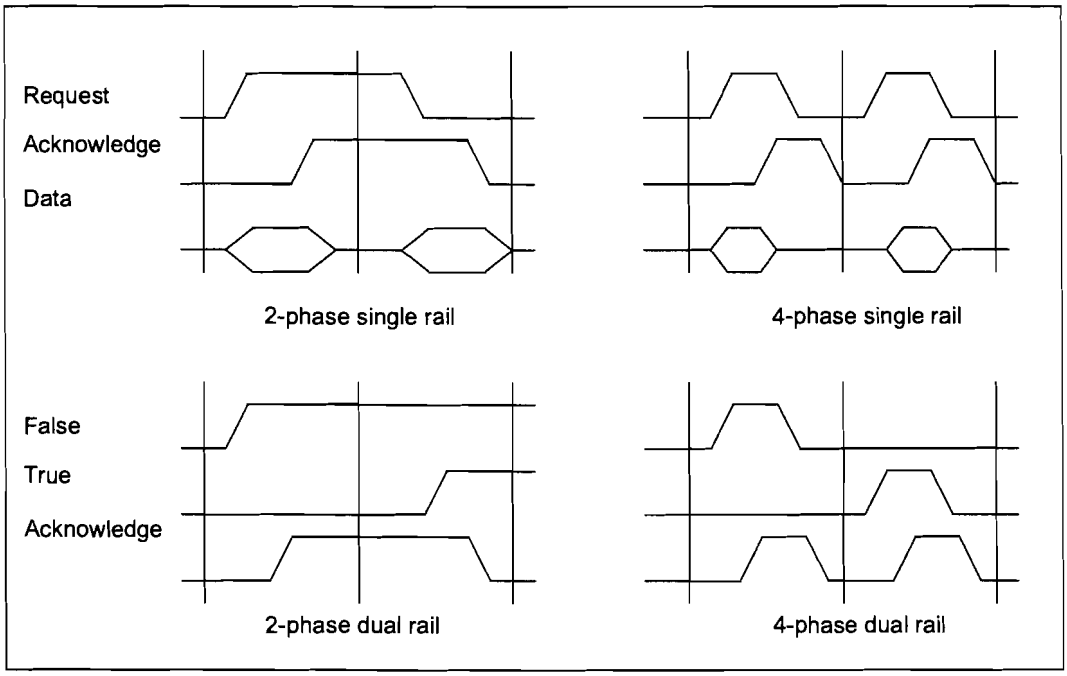
[Code Fragment 7] shows the channel communication between two parallel processes. The exclamation mark symbol denotes a send command, which evaluates expression *E* and then blocks until there is a parallel process willing to receive a value over channel *X*. The question mark symbol denotes a receive command, which block until there is a parallel process willing to send a value over channel *X*. When both commands are executing, variable *V* becomes equal to the value of *E* and both commands finish (see [7]).

```
( ... X!E ... ) || ( ... X?V ... )
```

[Code Fragment 7] Channel communication

2.2.2. Handshake channels

Handshake channels [5] synchronize two handshake components. The handshake protocol is implemented using the request and acknowledge signals with or without data transfer between the handshake components. Handshake components can be either active or passive. This is the difference between a component which initiates the handshake or which acknowledges the handshake. Handshake channels can be implemented in various ways. There can be a 2-phase signalling scheme or a 4-phase signalling scheme. Alternately there can be a single rail implementation or a double rail implementation.
The differences between the four implementations are show in [Figure 3] and explained below.



[Figure 3] Handshake protocols

In the 2-phase handshake protocol all transitions are functional, that is, each request followed by an acknowledgement constitutes a complete handshake. The 4-phase protocol involves four sequential events.
One 4-phase handshake consists of an up handshake followed by a down handshake and refers to the number of communications: (1) the sender issues data and sets request high, (2) the receiver absorbs the data and sets acknowledge high, (3) the sender responds by taking request low (data is no longer valid) and (4) the receiver acknowledges this by taking acknowledge low.
In the single rail handshake protocol the data will be combined to a bus where the handshake between the active and passive partner exists of a separate request and acknowledge signals. These request and acknowledge wires can act in a 2- or 4-phase manner.

The dual rail implementation encodes the request signal into the data signals using two wires per bit of information that has to be communicated. One wire is for sending a logical "1" (or true), the other wire is for the logical "0" (or false). There is still an acknowledge wire to complete the handshake.

2.2.3. Handshake circuits

Handshake circuits form the intermediate architecture between Haste and the gate-level implementation as a VLSI circuit. A handshake circuit has a set of handshake components that corresponds with the syntax constructs in the Haste design language. These handshake components abstract from the actual gate-level implementation, and merely provide a specification in the form of a handshake protocol. There is a library of approximately 40 components, which is sufficient to map the input from a Haste design to a handshake circuit. Examples of handshake components are the sequential construct and the parallel construct from [Code Fragment 1] and [Code Fragment 2], respectively seen in the previous section. For an elaboration on handshake circuits refer to [5].

2.2.4. Library connection

The connection to a specific library is established via a mapping file that specifies how the required asynchronous elements are composed in terms of cells in a standard-cell library. This is the final step in the functional design flow and is done by *htmap*.

2.3. Structural design flow

The structural part of the design flow is where multiple designs can be combined into one, so to support design reuse and modular design. This part of the flow also prepares a design for placement and routing and includes the design-for-test transformation. An overview is given in [Figure 4] and the different steps are discussed next.

[Figure 4] Structural design flow overview

2.3.1. Linking

The Handshake Technology design flow (TiDE) supports modular design. With this modular design it is possible to link different Haste programs as well as other generated combinational blocks into one integrated block.

2.3.2. Design for test

The main roadblock toward successful widespread use of asynchronous circuits has long been the lack of a design-for-test solution of industrial quality. Handshake Technology makes the handshake circuit scan-testable by adding a synchronous-style scan chain to the netlist. This way the design can be tested in large systems in which only a part might be asynchronous.

2.3.3. Preparing the netlist

The last step in the structural design flow is preparing and generating the netlist, scripts and additional files in formats which can be used by third-party tools for layout and optimization process.

2.4. Physical design flow

In this phase of the design, all tools are standard third-party EDA tools. The operation of these tools is partly controlled by scripting that, based on previous Handshake Technology files, instructs the tools how to handle the clockless circuitry correctly. Below [Figure 5] gives a schematic overview of the physical design flow.

[Figure 5] Physical design flow overview

Parameters which are being quantified, measured, set and fine tuned in this phase are for example timing constraints and delay elements.

3. 80C51

The 80C51 is an 8-bit microcontroller from Intel with a so-called CISC (Complex Instruction Set Computer, refer [1]) instruction set. It is one of the most popular 8-bit microcontrollers. Handshake Solutions in 1995 therefore decided to also make an asynchronous implementation, which has since then been used in many applications, especially in the smartcard domain.

This chapter introduces the 80C51 in four steps.

1. The original Intel architecture is introduced, which basically defines the instruction set architecture and memory interfaces.
2. The asynchronous implementation by Hans van Gageldonk [2] is presented, the HT80C51-LP (Low Power). This design was optimized for low power, but did not take scan-test and speed into consideration.
3. Based on an analysis of the HT80C51 design, a new design has been created that is known as the HT80C51-LC, which is optimized for a lower-cost introduction of scan-test. This design will be discussed and serves as the starting point for this thesis work, where the goal is to improve on speed.
4. A detailed comparison is made to available high-speed 80C51s on the market today. Also other asynchronous implementations are included in the comparison.

The last section of this chapter will be dedicated to performance analysis for an 80C51 microcontroller core.

3.1. Synchronous 80C51 implementation

The 80C51 microcontroller system consists of several parts; the CPU, its memories and different peripheral blocks. The CPU runs in parallel and communicates with the memories and peripheral blocks where and when necessary. The focus in this master thesis is on the CPU only, as shown in [Figure 6].

The major microcontroller features are:

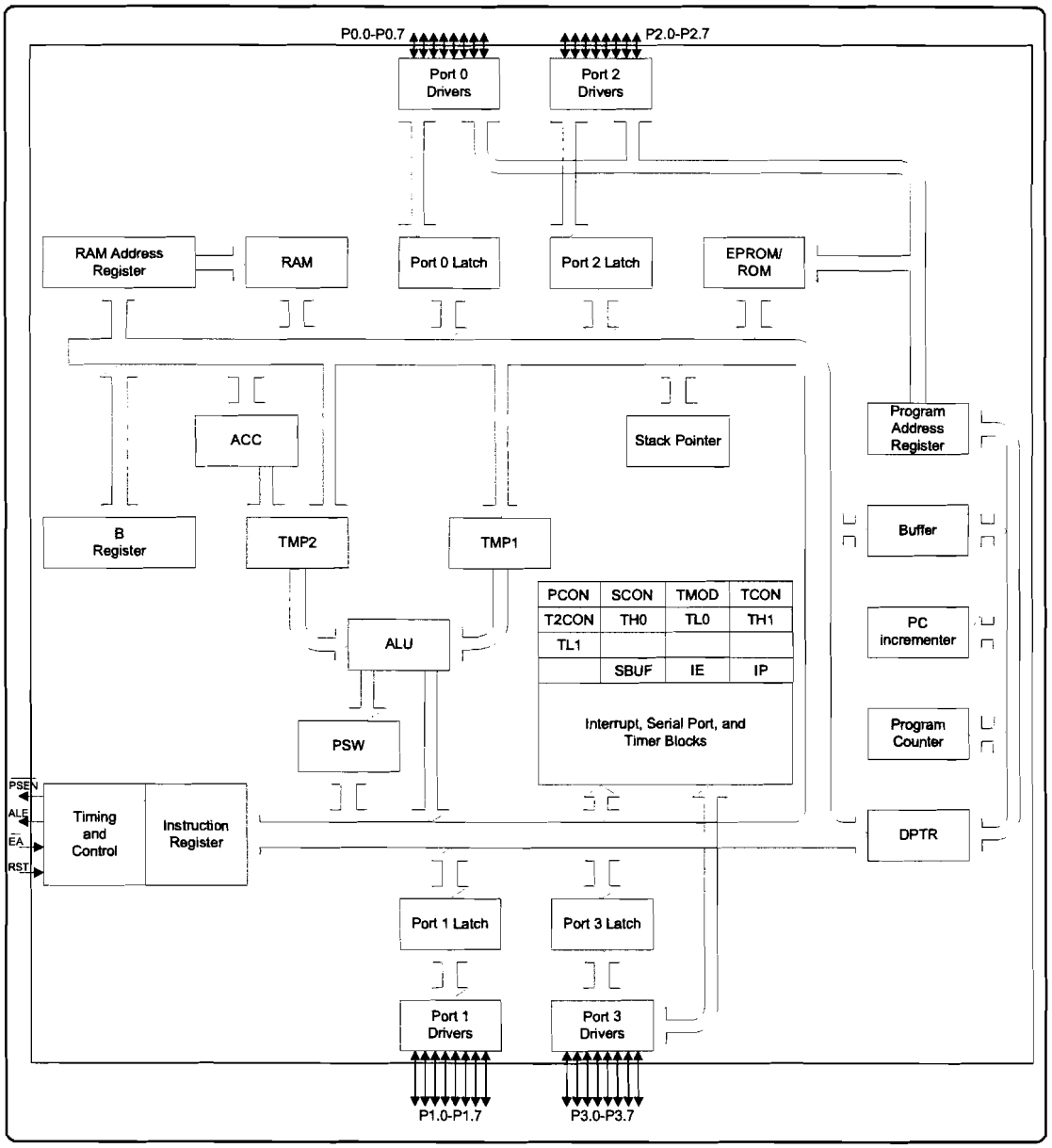
- An 8-bit central processing unit
- 64k Byte address space for the code/program memory
- 64k Byte address space for the external data memory

3.1.1. Memory organization

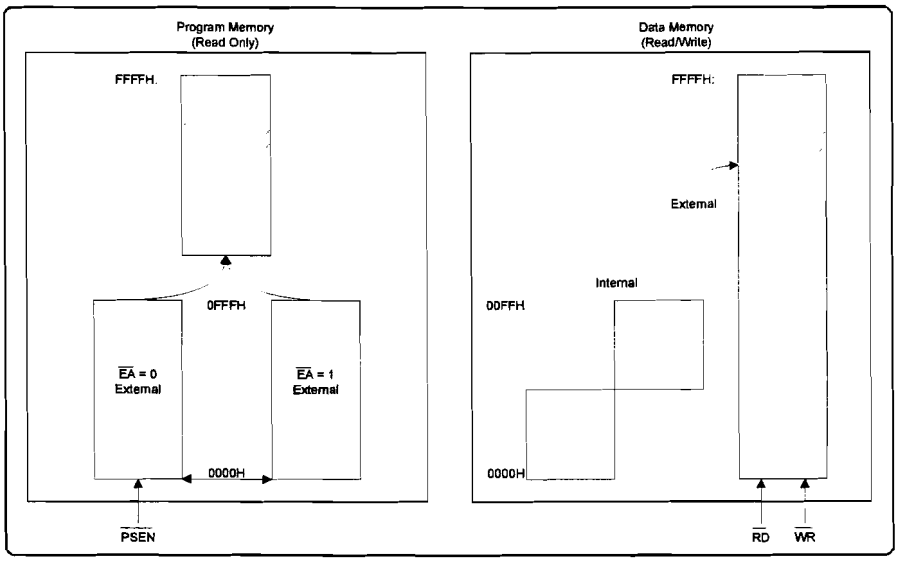
The 80C51 has separate address spaces for the code memory and data memory, known as the *Harvard* [1] architecture. The code memory can be up to 64 Kbyte long and may (partially) reside on-chip. A 16-bit program counter (PC) is the addressing mechanism.
The interrupt service routines occupy the ROM locations 03H through 32H. The start address after reset is 00H.

The data memory can consist of up to 64 Kbyte of off-chip RAM and is only accessed when an external move instruction is executed. The internal data memory is divided into three physically separate and distinct blocks: the lower 128 bytes of the RAM; the upper 128 bytes of the RAM; and the 128 byte special function register (SFR) area (see [Figure 7] and [Figure 8]). While the upper RAM area and the special function register area share the same address locations, they are accessed through different addressing modes.

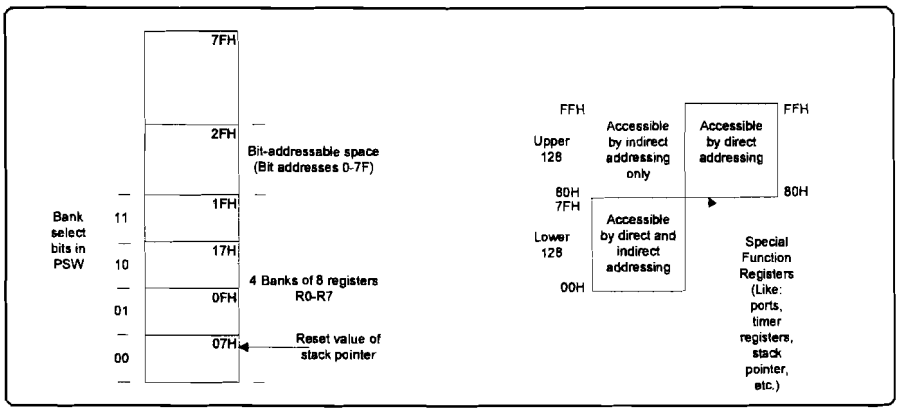
Four 8-register banks occupy locations 00H through 1FH in the lower RAM area. Only one of these banks is enabled at a time through a two-bit field in the program status word (PSW). The next sixteen bytes, locations 20H through 2FH, contain 128 bit addressable locations. The special function register area also has bit-addressable locations.



[Figure 6] Block diagram 80C51



[Figure 7] 80C51 memory structure



[Figure 8] Internal data memory

3.1.2. Special Function Register

The 80C51 microcontroller core can address 128 Special Function Registers (SFR). Five of them are related to the core. Some others are related to the peripherals of the core (like timers, watchdog, etc.), but not all addresses are occupied.
The five SFRs related to the core are the accumulator (ACC), B register, Program Status Word (PSW), Stack Pointer (SP) and the Data Pointer (DPL and DPH).

3.1.3. Addressing modes

The 80C51 incorporates several addressing mechanisms which each can access a part of the memory space. The five addressing modes and the associated memory spaces are the following:

1. Register addressing; has access to the eight working registers of the selected register bank. The accumulator, the B register, the data pointer, and the carry flag can also be addressed as registers.
2. Direct addressing; is the only method of accessing the special function registers. The lower 128 bytes of the internal RAM are also directly addressable.

3. Register-indirect addressing; uses the addressing contents as a pointer to a location in the internal RAM or the lower 256 bytes of the external data memory.
4. Immediate addressing; allows constants to be a part of the opcode instruction in the code memory.
5. Base-register plus index-register indirect addressing; allows a byte to be accessed from the data memory.

3.1.4. Instruction set

The 80C51 instruction set includes 255 instructions, 140 of which are single-byte, 91 two-byte, and 24 three-byte. The instruction opcode format consists of a function mnemonic followed by a "destination, source" operand field. This field specifies the data type and addressing method(s) to be used. The complete instruction set of the 80C51 family can be seen in Appendix A2. The instruction set is divided into five functional groups:

1. Data transfer; these operations perform the internal and external data byte movements.
2. Arithmetic operations; the 80C51 has four basic mathematical operations. Only 8-bit operations using unsigned arithmetic are supported directly.
3. Logical operations; the 80C51 performs basic logic operations on both bit and byte operands.
4. Boolean variable manipulation; operations on individual bits of registers.
5. Conditional program branching; all control transfer operations, some upon a specific condition, which disrupt the sequential program execution to continue at a non-sequential location in the code memory.

3.1.5. CPU timing

The synchronous architecture that implements the instruction set is shown in [Figure 6]. In this figure we see the registers, an ALU, the SFR-space and the four bidirectional ports. This is all designed around an *internal data bus* (IDB), from which all registers can read and some registers can write to. All communications between the registers use the IDB-bus, except for modifying the program counter (PC) which has a separate data bus. Having one bus for all these communications allows for only sequential execution of instructions. These executions take place in a number of steps, each of which communicates values from one register to another or do some calculations in the ALU. As the 80C51 has two data buses (the IDB-bus and PC bus) it is possible to do two steps in parallel in the instruction execution. It is, for example, possible to fetch a byte from the code memory while incrementing the PC.

The 80C51 instructions require a number of steps to execute. The instructions are executed with respect to a clocking scheme. Each instruction takes one, two or four machine cycles. Each machine cycle consists of six slots and each slot takes two clock cycles. Appendix A2 and A3 shows how many clock cycles each instruction needs to complete. The divide (DIV) and the multiply (MUL) are the only instructions that take four machine cycles to complete (due to the complexity of the execution of the instruction), the other ones take one or two machine cycles. A way to describe a general 80C51 instruction execution scheme is introduced in [2]. [Table 1] shows the executions steps which take place in each slot during a general instruction.

	One machine cycle					
	S1	S2	S3	S4	S5	S6
C1	ROM access	ACC -> T2	RAM access	ROM access	OP -> T1/T2	ALU -> destination

	One machine cycle					
	S1	S2	S3	S4	S5	S6
C2	ROM access	calculate jump address		PC incr.	OP -> T1/T2	ALU -> destination

[Table 1] General 80C51 instruction execution scheme

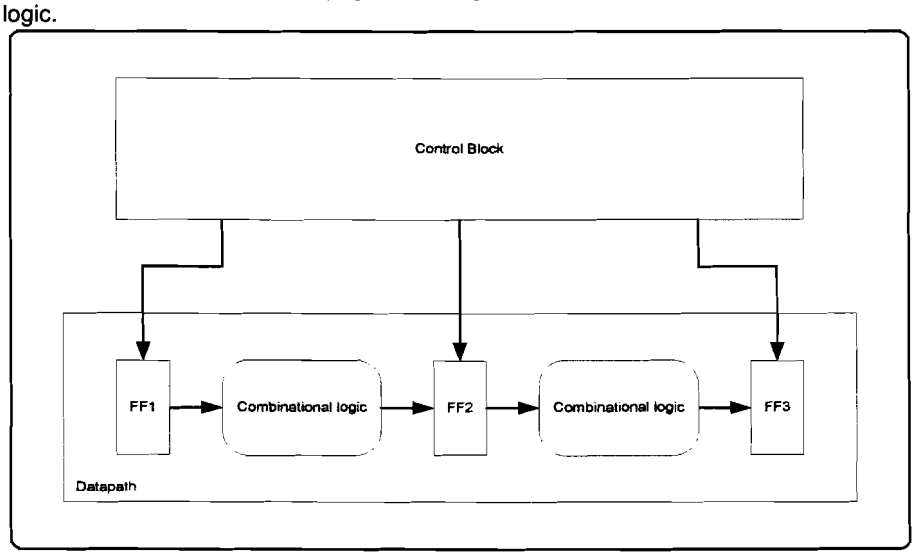
An instruction consists of one, two or three bytes; an opcode and two, one or zero arguments (operand addresses or immediate data). These bytes are read during the first and fourth slot of the first machine cycle and the first slot in the second machine cycle. Slot 2 of the first machine cycle copies the contents of the accumulator (ACC) into resister T2. For many instructions this action is redundant and this behaviour may be skipped. Slot 3 does a RAM access (which also includes access to one of the four register banks). Slot 5 and 6 of the first machine cycle take care of the ALU operation to be performed and the write-back to the destination register. Machine cycle 2 starts with another memory access, after which the jump instructions calculates their offset (slot 2 and 3). For two cycle non-jump instructions these actions are redundant. The fourth slot increments the program counter, and the 5th and 6th slot take care of an ALU operation. Approximately 30% of all slots contain redundant actions.

3.2. Low power HT80C51

The goal of the 80C51 microcontroller designed by Hans van Gageldonk (HT80C51-LP) [2] was to save power wherever possible. The second challenge was to keep the area of the circuit small. Therefore he decided to assume sequential execution of the instruction (fetch an instruction only after the execution of the previous instruction has finished), which makes it possible to reuse pieces of the datapath and logic blocks.

3.2.1. Datapath

An asynchronous implementation of the 80C51 (like the HT80C51-LP) is not much different from the implementation of a synchronous 80C51. Especially the datapath shows many similarities. There is still the need for registers, communication paths and arithmetic circuitry to perform the actual operations. However, the control of an asynchronous circuit is different from the centralized control (clock) of a synchronous circuit, see [Figure 9]. In an asynchronous design the clock does not determine when and where data is propagated through the datapath. Instead this is determined by control logic.



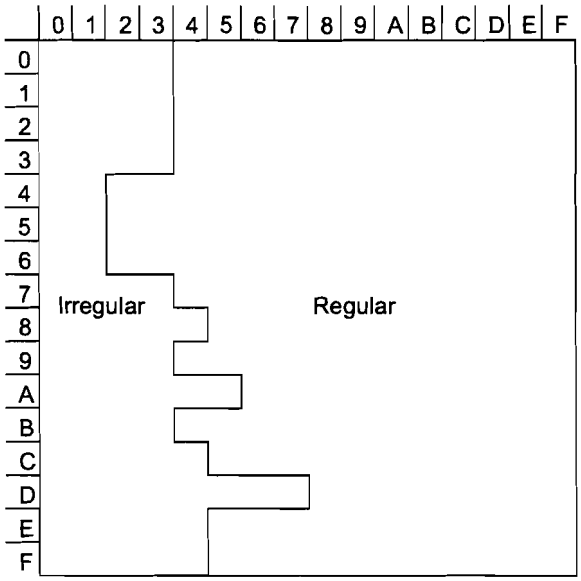
[Figure 9] Structure of a Handshake circuit

The 80C51 instruction code is analyzed to find overlap in the datapath, so that logic and communication paths can be shared where possible. The datapath of the HT80C51-LP implementation is based on a hybrid of point-to-point and bus communication. This means that a general bus is implemented for general communication between registers and arithmetic circuitry, like the IDB-bus (see 3.1.5) in the synchronous design.
The main advantage of the bus communication is that the circuitry is small. Every communication can be sent through this bus. Disadvantages are that it is slow, consumes more energy and limits parallelism.

In the datapath, communication between two registers always takes place in two steps via the bus: first the source register is copied to the bus; then the value of the bus is transferred to the destination register. Due to the latter disadvantage HT80C51-LP's implementation is a hybrid; a point-to-point communication is added for the most frequently used communications paths. These direct point-to-point communication paths bypass the IDB-bus. A few bypasses are introduced on the most frequently used paths, resulting in a circuit which is only marginal larger than a full bus implementation. But because of the frequency with which these bypasses are used, it also results in a fast and energy-efficient circuit.

3.2.2. Control structure

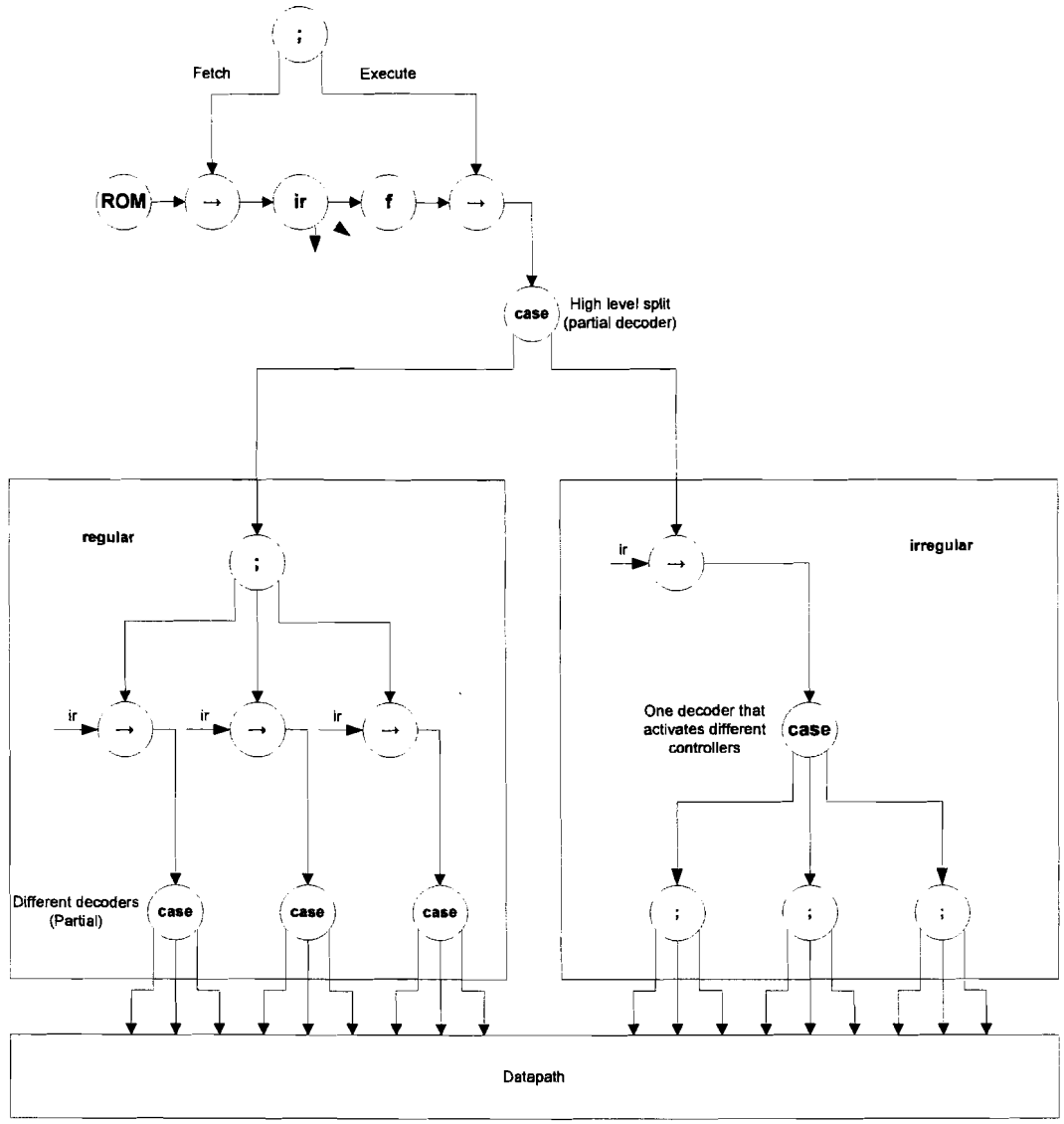
The control structure controls the communication and sequence in the datapath. The HT80C51-LP asynchronous implementation follows a sequential execution of instructions; first fetching an instruction from the code memory and then executing it. The execution of an instruction consist of two major steps: decoding of the instruction and executing the decoded instruction. The decoding of the execution also takes place in a hybrid way. First the instructions are separated in regular and irregular instructions according [Table 2]. The regular part is as large as possible to exploit the regularity in instruction execution as much as possible. The regular part itself is further being decoded in a distributed way. A lot of overlap in regular instructions demand the same actions and can use the same logic. The unique part of the regular instructions is then being executed individually. The irregular instructions are being decoded in a centralized way. First the complete irregular instructions are being decoded and then the appropriate action is being taken for each instruction. Also an overlap in the irregular part of the instruction set with similar irregular instructions is being clustered.
The disadvantage is that this design is constructed such that, after the instruction fetch, the control flow is highly dependant on the actual instruction, see [Figure 10].



[Table 2] Regular and irregular part of the 80C51 instruction set

[Figure 10] shows that almost every instruction (with its sequence of tasks) receives its own control flow. Because these different control flows contain quite some overlap in tasks (accessing code memory, accessing data memory, ALU operations, updating registers, updating the PC, etc.), the functions, procedures and variables that are responsible for these tasks are shared. This immediately creates another problem; sharing functions and procedures introduces an amount of glue logic. A logic block which is shared needs to be driven by the according tasks. Hereby mixers, multiplexers, demultiplexers and other logic are introduced to "glue" this together.
All this glue logic needs to be controlled by extra control logic. Also Design for Test (DFT) is very expensive due to the high control overhead.

Picture a handshake circuit that repeatedly executes a sequence of operations that consist of first fetching the instruction opcode and then splitting the flow of control into a large number of alternatives. All alternatives which merge back together and split up again several times in order to share a number of tasks, before the final merge that will eventually bring us back to the start of the repetition. This problem is called "The Sharing Pitfall" by [6]. Note that high control overhead is not only bad for area, but also for speed and energy.



[Figure 10] Handshake circuit for 80C51 control: hybrid decoding scheme

3.3. Low cost HT80C51

The low cost HT80C51 (also know as HT80C51-LC) consists of a single process. It leads through five sequential phases of executing instructions: fetch phase, decode phase, read phase, execute phase and the write phase. After the write phase it will start again with the fetch phase.

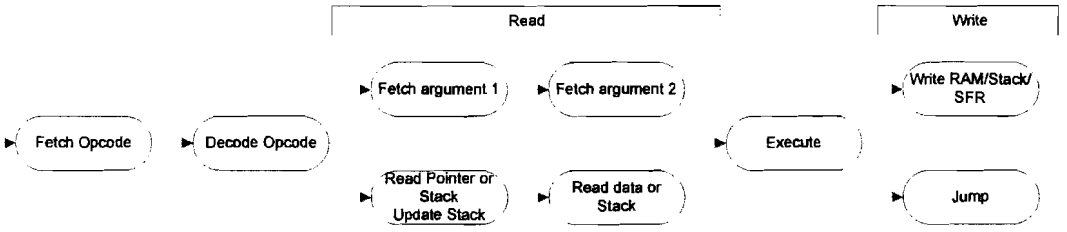
In the fetch phase the opcode is fetched from the ROM. First the code memory is addressed by the PC. After the ROM is addressed the instruction is being fetched from the ROM. Finally the PC is incremented.

The opcode (instruction) fetched by the previous phase is being decoded into proper condition signals, called EB (Execution Bits). This task is used to enable and configure the remaining tasks that are needed for the execution of the instruction.

The read phase makes sure that all arguments, if needed (denoted by the EB), are being fetched. This can be a single or double code memory fetch and/or a single or double data memory fetch.
The instructions are now ready to be executed by the next phase. The execute phase must be able to perform the operations specified in section 3.1.4 .

The last phase, write back, writes the executed data back to the data memory (SFR, internal memory, external memory). Also in the case of a branch the new PC is being calculated and stored.

The phases specified above are conditionally executed in this order, which allows a correct mapping of instructions. Based on data dependencies (a task must produce certain data before a second task is able to process that data), some tasks have to be sequenced in time, while others can be parallelized. For the HT80C51-LC the resulting ordering is as follows:



[Figure 11] HT80C51-LC data flow chart

Each of these phases takes a certain period of time. The total amount of time spent in all five phases is the instruction time. In order to increase the performance of the HT80C51-LC the instruction time needs to be decreased. The HT80C51-LC will be analyzed in such a way that the structure of the design is clear and ideas are being developed to decrease the instruction time of the HT80C51-LC.

3.3.1. Datapath

Again, the first step of the instruction is straightforward: fetching the instruction from the code memory at the correct address (PC). Then the instruction is decoded in a number of Execution Bits (EB). Some instructions contain overlap, e.g. the Execution Bits for an INC or a DEC instruction are more or less the same. So these two instructions share most of the datapath and the control structure. Although the vision of the HT80C51-LC design is not to share logic, for some specific blocks it is much cheaper to share and it will not reflect on speed.

Although the HT80C51-LC has a Harvard structure like the synchronous 80C51, an access to the external data memory never occurs during an access to the code memory. This allows for an un-

arbitrated wrapper around the code and data memory bus that repartitions these buses into one code and external data memory bus.

As seen in the Appendix A2 the instructions of the HT80C51-LC are one, two or three bytes in length. These bytes are all sequentially being read from the ROM. Also the RAM can be accessed three times, two read accesses and a write access as seen in Appendix A3.

A typical HT80C51 instruction will follow the next execution scheme:

Fetch	Decode	Read		Execute	Write
ROM access opcode	Decode the opcode	ROM access arg1 RAM access	ROM access arg2 RAM access	Mul/Div ALU	RAM access write

[Table 3] General HT80C51 instruction execution scheme

Some of the steps in [Table 3] are redundant for some instructions. As example; instruction *RR A* will only need the Fetch phase (for the Opcode), Decode phase and the Execute phase. The complete instruction execution time and memory utilization are specified in Appendix A3.

Fetch	Decode	Read	Execute	Write
ROM access opcode	Decode the opcode	Skip	ALU	Skip

[Table 4] *RR A* execution scheme

As seen a double RAM read is possible in the HT80C51, because the design choice was made to have the R0-R7 registers in the addressable RAM and not in internal registers. The advantage of this implementation is a more transparent control flow and a smaller area.
When the instruction *MOV @Ri, direct* will be executed, it will follow the next execution scheme:

Fetch	Decode	Read		Execute	Write
ROM access opcode	Decode the opcode	ROM access arg1 RAM access	RAM access	ALU	RAM access write

[Table 5] *MOV @Ri, direct* execution scheme

The first ROM access is the opcode fetch, after this the opcode is being decoded. Then the first argument of the instruction is loaded (direct address), and parallel to this the address of @Ri is being determined from the data memory. After this the direct address is available to load the data from the data memory. When everything is gathered the ALU will be started and eventually the data is written back into the data memory.

The disadvantage of having the register bank (R0-R7) in the data memory becomes more clear with the instruction *INC @Ri* , which will take an extra RAM read access. The extra RAM access is needed for loading the Ri register and compute the address for the RAM.

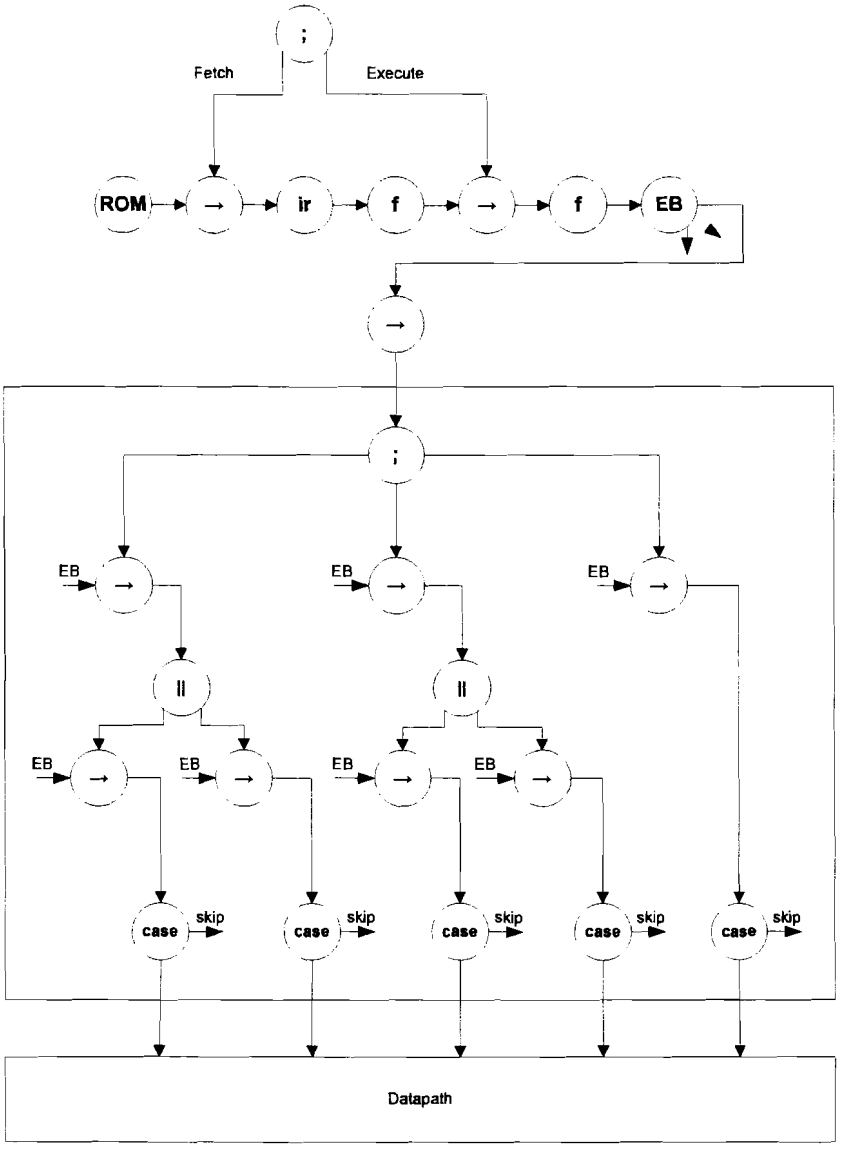
Fetch	Decode	Read		Execute	Write
ROM access opcode	Decode the opcode	RAM access	RAM access	ALU	RAM access write

[Table 6] *INC @Ri* execution scheme

It is clear from these three examples that the 80C51 has a complex instruction set. The different (255) instructions all take different ways through the execution scheme with different effects or skipping for each stage.

3.3.2. Control structure

The HT80C51-LC implementation of the control structure is designed to prevent the extensive amount of glue logic which would increase the complexity of the control structure. To prevent glue logic there should be no additional resource invocation beyond the first one. This is only possible when the control structure is not split up when several of the resulting paths invocate a common resource. The tasks in a sequence of tasks are executed conditionally in case not all sequences contain those tasks. In other words, the executions of an instruction amounts to decoding what tasks correspond with this instruction and skipping the execution of the other tasks in the sequence of tasks the design contains. This way the control structure is not split up like the HT80C51-LP implementation (see 3.2) but is organised as shown in [Figure 12].



[Figure 12] HT80C51-LC handshake circuit for control structure

```
Decode =
[
  << EB0, EB1, EB2, …, EBX >> := << EBAdd, EBCall11, EBDecSP, …, EBIncPC >>
]
```

[Code Fragment 8] The Execution bits decoded by the decoder

After the decode phase it is clear what the other phases need to do because all the proper execution signals, the EB, are decoded (see [Code Fragment 8]). So tasks after the decode phase execute a sequence of conditional tasks. The read phase will conditionally read the other arguments (instruction arguments) and will conditionally perform RAM accesses.

```
ReadData =
[
  If EB_Fetch1  then handleFetch1()  fi ||
  If EB_AuxRead then handleAuxRead() fi ;
  If EB_Fetch2  then handleFetch2()  fi ||
  If EB_OpRead  then handleOpRead()  fi ;
]
```

[Code Fragment 9] The Execution bits used in the read phase

The same procedure holds for the other phases that the decoder already decoded the Execution Bits and therefore it will be straightforward which part of the CPU will do the execution. For example the DivideMultiply part, or the ALU which can *Add, Rotate, Compare, Move*, etc. bits or bytes.

```
Execute =
[
  If EB_DivMul  then handleDivMul()  fi ||
  If EB_ALU     then handleAlu()     fi ||
  If EB_Stack   then handleStack()   fi ;
]
```

[Code Fragment 10] The Execution bits used in the execute phase

The last part is the write back phase. This phase will write back the calculated data from the ALU to the SFR register, data memory or internal registers. Also a branch address which is calculated in the executed phase can stored in the PC.

```
Write Data =
[
  If EB_Write   then handleWrite     fi ||
  If EB_Jump    then handleJump      fi ;
]
```

[Code Fragment 11] The Execution bits used in the write phase

3.4. Comparison high-speed 80C51 microcontrollers

This section describes in more detail the existing synchronous and asynchronous 80C51 microcontroller cores. To make an adequate comparison between 80C51 designs we need to examine the current synchronous and asynchronous implementations.

The 80C51 is the world's most famous and most used microcontroller. Many IP builders and IC manufacturers have implemented their version of the 80C51 microcontroller.
Because of the limited documentation available in public (e.g. datasheets, user guides, websites, papers, etc.) it is difficult to determine the architecture of the different microcontrollers.

There are thousands of different 80C51's made and sold by more than 60 different manufacturers. A lot of these manufacturers still implement the original architecture of the 80C51 designed by Intel. This means 12 clock cycles per machine cycle and some instructions use more than one machine cycle (refer [3]). Some implementations of the 80C51 microcontroller use higher clock speeds than the original 12 MHz. A lot of the IC manufactures implement high-speed versions of the 80C51 microcontroller. Most of the time this means that the original 12 clock cycles per machine cycles needed is reduced to 6 clock cycles per machine cycle. This means a two times higher instruction rate (MIPS) than the original core with the more or less the original architecture. The original 80C51 design was based on latches which needed to be clocked at a double clock speed. When replacing these latches by flip-flops there is no need for 12 clock cycles per machine cycle, but only 6. This is a common speed up by many manufacturers.

Several synchronous and asynchronous implementations (which are considered the top of the line regarding speed performance) are described below.
To make an adequate comparison with different 80C51 microcontrollers implemented with both synchronous and asynchronous techniques the performance will be give in Million Instructions Per Second (MIPS). This measurement is good for comparing the same instruction set in the same technology. Some instruction sets make it possible to express a program in fewer instructions than other instruction sets. On the other hand, some instruction sets contain more powerful instructions than others. This makes it difficult to compare the MIPS metric for one type of microcontroller to another with a different instruction set.

The standard 80C51 runs at a frequency of 12 MHz and uses 12, 24 or 48 clock cycles for one instruction. The performance will be somewhere between 0.25 and 1 MIPS depending on the program being executed, but never better.
It is often not clear in which environment, conditions, libraries and technologies these implementations are benchmarked, so the numbers gives are mostly a guide and not exact figures. However the architecture and the general performance can give us a good guiding map for the decisions that need to be made.

	Manu-facturer	Type	Async/ Sync	Pipeline #	Techno-logy µm	Power · mW	Area # trans	Through put MIPS
1	NXP [13]	H8051	Sync	No	0.35 3.3V	40	-	4 (peak)
		P89xxx	Sync	-	-	-	-	33 (peak)
2	Intel [14]	MCS51	Sync	3 stages	-	-	-	12 (peak)
3	Cast [15]	R8051XC	Sync	-	0.13	-	188.640	256 (peak)
4	Dolphin [16]	Flip8051-Cyclone	Sync	Yes	-	-	-	3.75 (average)

5	Silicon Labs [17]	CIP51	Sync	Yes	0.35	-	-	23 (peak)
6	Dallas – Maxim [18]	DS89C4xx	Sync	Yes	0.35 1.1V	55	-	33 (peak)
7	Atmel [19]	AT89	Sync	No	-	-	-	3.3 (peak)
8	Chipcon [20]	Texas	Sync	-	3.6V	-	-	8 (peak)
9	Digital Core Design [21]	DP805x	Sync	Yes	-	-	-	85 (peak)
10	CalTech [11]	Lutonium	Async	Yes	0.18 1.8V	100	-	200 (peak)
11	Chungbuk University [9]	A8051	Async	5 stages	0.35 3.3V	46	104.000	76 (average)
				No	0.35 3.3V	-	-	36 (average)
12	Philips [10]	HT80C51 – LP	Async	No	0.50 3.3V	9	39.174	4 (average)
13	Handshake Solutions [8]	HT80C51 – LC	Async	No	0.14 1.8V	0.7	30.820	8.9 (worst case)

Note: All the numbers found on the references mentioned

[Table 7] Detailed comparison 80C51's

1. NXP semiconductors (founded by Philips) offers a wide range of 80C51 microcontrollers including the original 80C51 with Intel specification. This original version has a machine cycle of 12 clock cycles like previously discussed in section 3.1. This version of the NXP microcontroller is often the benchmark for other manufactures.
   NXP also designed high-speed versions of the 80C51. They execute single-cycle instructions with an input clock of 33 MHz. Not all instructions are single-cycle, so the maximum throughput is 33 MIPS.

2. Intel the original inventor offers the 80C51 microcontroller with the same basic functionality as the original, but with improved performance regarding speed and power. Intel offers 24 MHz input clock frequency 80C51 based microcontrollers with two clock cycles per instruction. The microcontroller is implemented using typical pipelining techniques, and is built around a three-stage pipeline. The pipeline stages are instruction fetch or decode, address generation or data fetch and execution or write back. The three-stage pipelined implementation offers the best trade-off between performance and design complexity, according to Intel (refer to [14]). The code memory bus is 16 bits wide while the data memory bus remains 8 bit like the original 80C51 microcontroller.

3. CAST's version of the 80C51 microcontroller executes operations on an average of eight times faster than Intel's original design at the same clock speed due to the single clock cycle per machine cycle implementation. Some single byte instructions can execute in one clock cycle. CAST's architecture is strictly synchronously designed. Evatronix uses this core in their single-chip applications of the 80C51. At Evatronix the core has a performance of 45 MHz on a 0.5-micron process. [Table 8] illustrates the speed advantages of the R8051 over the standard 80C51. A speed advantage of 12 means that the R8051 performs the same instruction 12 times faster than the original 80C51. The average of speed advantage is 8x, however, the actual speed improvement observed in any system will depend on the instruction mix.

The internal and external data memory interfaces are 8 bits wide as well as the external code memory interface.

Speed advantage	Number of instructions	Number of opcodes
24	1	1
12	27	83
9.3	2	2
8	16	38
6	44	89
4.8	1	2
4	18	31
3	2	9
Average: 8.0	Sum: 111	Sum: 255

[Table 8] CAST's R8051 speed advantages

4. Dolphin's implementation of the 80C51 microcontroller runs at a 50 MHz clock input on a Dhrystone testbench at 3.75 MIPS. This 80C51 implementation is a pipelined architecture that reduces the number of clocks per instruction. The advanced version of Dolphin's 80C51 (with the enriched C51 instruction set) features a 4-stage pipeline.

5. CIP51 designed by Silicon Laboratories (Cygnal) in the range of C8051F0xx has several enhancements inside and outside the CIP51 core to improve its overall performance. The CIP51 is fully compatible with the MCS-51 instruction set (comparable to the 80C51). The CIP51 employs a pipelined architecture that increases its instruction throughput. The CIP51 executes more than 70% of its instructions in one or two clock cycles instead of the twelve clock cycles of the standard 80C51 (refer [17]). With the CIP51's maximum system clock at 25 MHz, it has a peak throughput of 25 MIPS.
   Due to the pipelined architecture of the CIP51, most instructions execute in the same number of clock cycles as there are program bytes in the instruction. Conditional branch instructions take one less clock cycle to complete when the branch is not taken as opposed to when the branch is taken.

Instructions	26	50	5	14	7	3	1	2	1
Clocks to Execute	1	2	2 / 3	3	3 / 4	4	4 / 5	5	8

[Table 9] Number of instructions with clocks to execute, CIP51

6. The Dallas/Maxim DS89C4xx family also features a pipelined architecture. The DS89C4xx is 100% functionally compatible with the original Intel 80C51 microcontroller, but allows operation at a higher clock frequency. This core does not have the wasted memory cycles that are present in a standard 80C51. A conventional 80C51 generates machine cycles using the clock frequency divided by 12. The same machine cycle takes one clock cycle in the DS89C4xx. Thus, the fastest instructions execute 12 times faster for the same crystal frequency. At one cycle machine instruction this core has a peak throughput of 33 MIPS at a system clock of 33 MHz.

7. Atmel AT89, Atmel offers an 80C51 family which runs at 20 MHz with 6 clock cycles per machine cycle. This microcontroller offers the same functionality as the original 80C51 by Intel. The architecture of the AT69 is a non-pipelined version with a maximum performance of 3.3 MIPS. The code and data memory are accessed through a dedicated 8 bit wide bus.

8. Chipcon uses cores from Texas Instruments. These high-speed 80C51 cores use a 32 MHz clock input to achieve a maximum throughput of 8 MIPS. They achieve this by a 4 clock cycle per machine cycle rate. Not all instructions execute in a single machine cycle. As can see in

the table below, for every byte fetch there is a need for an extra instruction cycle. A single byte instruction can be read from the code memory, read the appropriate byte from the data memory, being calculated and written back to the data memory all within the same machine cycle (see instruction *INC @Ri*).
It seems that this architecture executes several processes (ROM fetch, decoding, RAM fetch, Execution and Write back) in sequence before the instruction is completed. It is likely that the R0..R7 reside locally instead of in the data memory. This can be concluded because of the single cycle with multi data memory accesses instruction like the *INC @Ri*.

Mnemonic	Description	Code	Bytes	Cycles
ADD A,Rn	Add register to accumulator	0x28 – 0x2F	1	1
ADD A,direct	Add direct byte to accumulator	0x25	2	2
ADD A,@Ri	Add indirect RAM to accumulator	0x26 – 0x27	1	1
INC Rn	Increment register	0x08 – 0x0F	1	1
INC direct	Increment direct RAM	0x05	2	2
INC @Ri	Increment indirect RAM	0x06 – 0x07	1	1

[Table 10] Chipcon's Texas Instruction Set

9. Digital Core Design DP805x pipelined microcontroller is fully compatible with the original 80C51 instruction set. Every machine cycle is equal to one clock cycle. But not all single byte instructions execute in a single clock cycle. This depends on which operands are needed to execute the instruction; reading from the data memory interface automatically adds an extra clock cycle to the instruction time. Some instructions need up to 5 clock cycles. It is not clear what kind of architecture the DP805x has, but from [Table 11] some architecture choices can be derived.
The *INC Rn* and the *INC @Ri* need just the opcode. But the *INC @Ri* takes three cycles to execute. This is because the additional access to the memory space takes an extra cycle. Similarly for the extra byte fetch from the opcode takes place in the second cycle as can be derived from instructions *ADD A,direct* and *INC direct*. Each access to the memory takes place in a separate cycle.
From this it is clear that the R0..R7 registers are resided in the memory space and that the code and data memory are separately fetched in a single byte access.

Mnemonic	Description	Code	Bytes	Cycles
ADD A,Rn	Add register to accumulator	0x28 – 0x2F	1	1
ADD A,direct	Add direct byte to accumulator	0x25	2	2
ADD A,@Ri	Add indirect RAM to accumulator	0x26 – 0x27	1	2
INC Rn	Increment register	0x08 – 0x0F	1	2
INC direct	Increment direct RAM	0x05	2	3
INC @Ri	Increment indirect RAM	0x06 – 0x07	1	3

[Table 11] Digital Core Design DP805x Instruction Set

10. The Lutonium is CalTech's (California Institute of Technology) design for an asynchronous 80C51 architecture microcontroller. This is a pipelined asynchronous 80C51 microcontroller designed for low energy and high performance. In 0.18 µm CMOS, at nominal 1.8V, the expected performance is an impressive 200 MIPS.
The Lutonium includes a multi-stage pipeline and a fetch unit which fetches 16 bit of code from the code memory. To decrease the power consumption, the fetch unit is nonspeculative: the fetch unit only keeps filling the pipeline as long as it is known at the instructions are going to be executed.
Activity is localized as much as possible: special registers (SP, PSW, B, DPTR) have their own channels and function units instead of using the main buses and units.

Not all instructions run in one cycle, the *CALL* and *RET* instructions need two cycles to complete.
There are only estimated results known of the Lutonium, the designer of this microcontroller claims a 25x better performance regarding speed than the asynchronous design described at point 12, the HT80C51-LP in the same technology with similar conditions.

11. Chungbuk microcontroller A8051 is an asynchronous version of the 80C51 microcontroller. This microcontroller introduces a well-tuned 5 stage pipelined architecture. This microcontroller has some enhanced features to improve the system performance, like an instruction register to fetch a complete word from the code memory, branch prediction and other features.
    The instructions are regrouped in seven groups. These seven groups are divided by the execution scheme of the 80C51 instructions. Some instructions only need parts of the execution scheme (e.g. *NOP* instruction only needs to be fetched and decoded). This way the regrouped instructions (regrouped for the same machine cycles) use the same set of pipelined stages. Thus, each group acts like RISC instruction sets. Some instruction groups need to run a pipeline iteration multiple times (see [Table 12]) to fully complete the execution (e.g. *RET*, *CALL* instructions). The R0..R7 registers residing in the data memory.
    The designer of the A8051 microcontroller claims a 5x faster microcontroller than the asynchronous design described at point 12, the HT80C51-LP in the same technology with similar conditions.

Group	**1**	**2**	**3**	**4**	**5**	**6**	**7**
# of instruction	22	13	93	89	4	6	28
# of reduced stage	4	2	1	0	3	2	6
# of needed iteration	0				1		2

[Table 12] Chungbuk reduced execution stages of each group

12. The HT80C51-LP (Low Power) microcontroller designed (and discussed in [12]) by Philips is the first 80C51 microcontroller designed in an asynchronous architecture. This design was important to demonstrate the feasibility and the advantages of the low-power Tangram asynchronous design flow. The architecture of the microcontroller is described in detail in section 3.2. It is a non-pipelined sequential architecture, due to the low-power implementation the microcontroller is mainly designed to reuse pieces of datapath and logic blocks. This is a disadvantage because there is a need for glue logic to combine the mixers, multiplexers and other blocks which are necessary together.
    The HT80C51-LP is designed with low-power as main design goal and therefore this design is not optimized for speed. The HT80C51-LP runs at an average 4 MIPS rate in a CMOS 0.5 µm technology at 3.3V. If this number is scaled to our current technology (CMOS 0.14 at 1.8V in worst case scenario) it would run approximately 6 MIPS.

13. Handshake Solutions HT80C51-LC (Low Cost) microcontroller is based on the HT80C51-LP microcontroller. This design is discussed more in detail in section 3.3. The design goal of this microcontroller is low cost, but also performance and power are important. With an area of 7700 gates the performance is 8.9 MIPS (CMOS 0.14 at 1.8V worst case). The power consumption is below 70pJ/Instruction. This implementation of the 80C51 microcontroller is completely asynchronous and sequential. The R0..R7 registers are placed in the data memory to decrease the area overhead. The code and data memory are accessed by separate 8 bit buses.

It is complicated to compare all the designs previously described in this section. There are a lot of different types of architectures, not all public and known. Every design is tested and benchmarked with different types of programs, in different technology modes and processing corners with different technology libraries and performance grades. Therefore a huge difference in power consumption, area costs and speed performance between designs mapped onto CMOS 0.18 µm versus CMOS 0.35 µm, or designs run at 3.3V versus 1.8V, or worst case scenarios versus typical case scenarios, type of

memories, etc is possible. These kinds of differences between compilation and test conditions can easily make 500% difference in performance, area and power between mutual designs. So unless all the test, memories and benchmark variables are available it is complicated to make a fair comparison.

The two references we have in detail are the HT80C51-LP and the HT80C51-LC. The latter is where we are going to base our further analysis and numbers on for this graduation project. The first, HT80C51-LP is the core where some other microcontrollers reflect to. The Lutonium and Chungbuk 80C51 microcontroller are benchmarked to the HT80C51-LP. According to these benchmarks the Lutonium runs approximately 25x faster (refer to [22]) and the A8051 5x faster than the HT80C51-LP microcontroller (refer to [23]), but once more, the Lutonium performance is an estimate and both the Lutonium as the Chungbuk implementations are scientific developments. This means that the technology libraries and other test conditions can be more optimized and not comparable to the conditions and variables for which the HT80C51 microcontrollers are benchmarked.

3.5. Theoretical performance analysis

As the HT80C51-LC (analyzed in section 3.3) is a sequential design, although some tasks are done in parallel, it is this design we take as starting point for this thesis work. Therefore the design is not only analyzed but also measured in terms of speed, area and power. For reference this design is also compared to synchronous 80C51 architectures and the original HT80C51-LP design. The HT80C51-LC is designed in Haste and completely compiled with the Handshake Technology design flow into a standard-cell CMOS 0.14 µm netlist.
The maximum speed of the asynchronous microcontroller depends on which instruction is being executed. E.g. an *ADD* (addition) instruction executes faster than a *MUL* (multiply) instruction. Not because of the amount of memory accesses, but because of the control and data logic of the *MUL* instruction is far more complicated than the logic of an *ADD* instruction.

To determine the execution speed we take the average of a benchmark program, which executes every instruction at least once. The total time of the benchmark program divided by the number of instructions is the average time it takes to execute an instruction. Additionally every single instruction is analyzed with a benchmark program to see what tasks of execution it takes and in what amount of time. [Table 13] shows the total execution time of every single instruction, number of memory accesses and percentage it takes of the total execution time to do a specific memory access. All 255 instructions of the 80C51 instruction set are divided in 18 groups (for division of the groups see Appendix A5) by their number of code memory accesses and number of data memory accesses.

The memory delay is set at 5 ns for both the data as code memory in simulation mode. This is a fast, but realistic delay for on-chip memories in a CMOS 0.14 µm process.
Unfortunately a memory access has overhead in the form of address calculation, overhead by control logic and others. This overhead is taken into account with the ROM/RAM times + OH (overhead). The ROM/RAM times are calculated with 5 ns memory delay.

Instr. Group	ROM fetch #	RAM read #	RAM write #	Total instr time ns	ROM time + OH %	RAM time + OH %	ROM time %	RAM time %
1	1	0	0	66-202	10 – 32	0	2 – 7	0
2	1	0	1	87	24	24	5	5
3	1	1	0	104	20	20	4	4
4	1	1	1	104-123	17 – 20	34 – 40	4 – 5	8 – 10
5	1	2	0	120	17	35	4	8
6	1	2	1	134-139	15 – 16	45 – 47	4	10 – 11
7	1	3	0	158	13	39	3	9
8	2	0	0	88-93	45 – 47	0	10 – 11	0
9	2	0	1	107	39	19	9	4
10	2	0	2	137	30	30	7	7
11	2	1	0	120-123	34 – 35	17 – 18	8	4
12	2	1	1	107-145	29 – 39	29 – 39	6 – 9	6 – 9
13	3	0	0	107-132	47 – 59	0	11 – 14	0
14	3	0	1	130	48	16	11	3
15	3	0	2	156	40	26	9	6
16	3	1	0	115-120	52 – 55	17 – 18	12 – 13	4
17	3	1	1	137	46	30	10	7
18	3	2	0	120	52	35	12	8

Note: OH: Overhead

[Table 13] HT80C51-LC memory usage

As shown in [Table 14], the HT80C51-LC processor is about the same speed as other non-pipelined designs, when compared after compensation for technology differences using a scaling factor (all speeds are scaled to worst-case CMOS 0.14 µm process).
From the analysis in section 3.3, [Table 13] and [Table 14], it is clear that accesses to both data and code memories and the actual execution phase of an instruction are the bottleneck. The code memory and the data memory only have one addressing port and therefore can only be addressed sequentially. For example (see Instruction Group 18 from [Table 13]) when an instruction is executed which needs the Opcode, 2 ROM accesses and 2 RAM accesses before the instruction itself can be executed, 87.5% of the total instruction time is spent on memory accesses.
This time includes the total overhead by control and logic which is needed to do an actual memory access.

Version	8051 [3]	CIP51 [17]	HT80C51 [2]	HT80C51-LC
Type	Sync @48 MHz	Sync @50 MHz	Async	Async
Architecture	Non-pipelined	Pipelined	Non-pipelined	Non-pipelined
MIPS	5	26	8	8.9

[Table 14] Speed comparison with other versions

The access to the memory is done in two steps: addressing and receiving (or writing) the data. Before an operand can be fetched its address should be calculated. For fetching an operand the Program Counter (PC) will hold the memory address. A more detailed description of the memory access is described in chapter 5.9.1.
The goal will be to keep the code memory occupied as close to 100% as possible. The other tasks in the instruction scheme, like decoding, data accesses, instruction executing, will be done in parallel and therefore pipelining [1] needs to be introduced. It is not possible to get a full 100% memory usage, because of jump and other branch instructions. Conditional branches require being decoded and executed before the correct branch can be taken.

When executing a dedicated testbench with all possible 80C51 instructions with a total of 297 code memory accesses. This means that when the code memory is 100% occupied the minimum time needed to execute this testbench will be 6237 ns (160 instructions with a total of 297 memory accesses x 21 ns for each memory access). The time it takes to run this testbench on the HT80C51 is 17800 ns. So the theoretical maximum speed improvement in this technology (CMOS 0.14 µm library) will be 2.8 times faster.

When the total instruction scheme will improve (e.g. optimizing the Haste code, optimizing the architecture of the HT80C51, etc.) and the accesses to the memory will get occupied for 100%, we need to improve the memory access process described in 5.9.1. At the moment a total memory access will take 21 ns (CMOS 0.14 µm technology), while the actual access to the memory takes a total of 5 ns. The overhead of 16 ns is due to: calculation of the memory address, the addressing of the memory, the actual data transfer and control overhead. Theoretically it is possible to optimize the memory access such that it only takes the time to do the actual access of 5 ns plus a minimal setup time required by the logic and control. Here it is possible to find an improvement for speed by a factor of 4.2 (21/5 ns).

	Instruction	**Code memory access**	**Program execution time**	**Redundant execution time**
	#	#	ns	%
HT80C51	160	297	17800	91,6
Optimal High Speed HT80C51	160	297	6237	76,2
Optimal memory access HT80C51	160	297	1485	0

[Table 15] Optimizing memory accesses in the HT80C51

When optimizing the Haste code or the HT80C51 architecture it is important that all tasks will fit in the time slot it takes to do a memory access. We need to split up all the tasks to small enough subtasks; this can be done by structures like pipelining. Pipelining will make the control structure and datapath structure more complex and therefore introduce more area.
Other types of optimization are remapping different parts of the architecture (like residing parts in the memory or locally in the core), optimizing executions of instructions (faster adders, shifters, etc.), using mechanisms like data forwarding, etc.

The next chapters will describe such mechanisms to speed up the HT80C51 microcontroller.

4. Initial speed up of the HT80C51

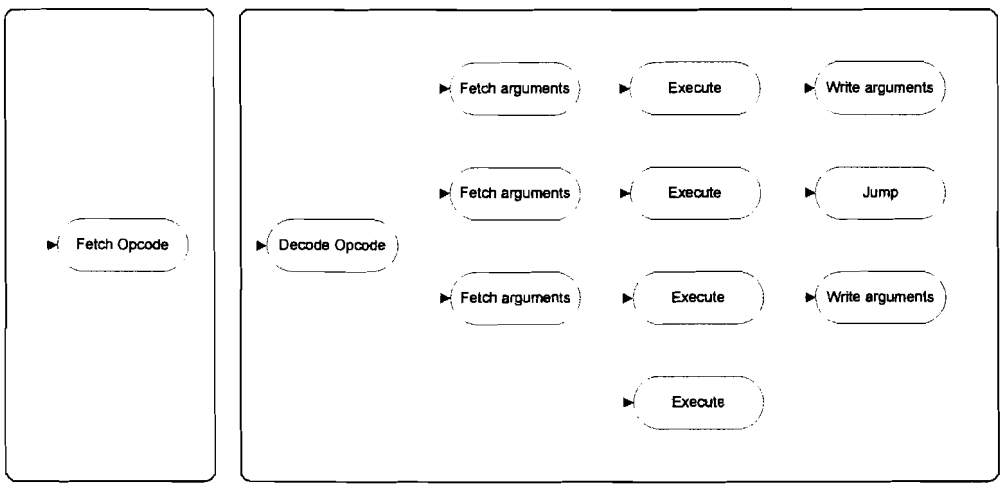
This chapter presents an initial speed up of the HT80C51 microcontroller, the HT80C51-SU (Speed Up). Section 4.1 starts by introducing the main architecture. It discusses the objects that make up the design, as well as their interaction with each other. After this, sections 4.2 and 4.3 discuss the datapath and the control structure of the HT80C51-SU, respectively. Finally section 4.4 concludes with some benchmark results and an approach to achieve better results.

4.1. Conceptual architecture

The goal of this design is to improve the HT80C51 performance without a major architecture change. Therefore no extreme form of pipelining is introduced, but there is a pre-fetch unit. Some non-branch instructions can already, after their last read access to the code memory, fetch a new instruction opcode while the current instruction is being executed. This form of pipelining, pre-fetching, does not require a lot of additional control logic.

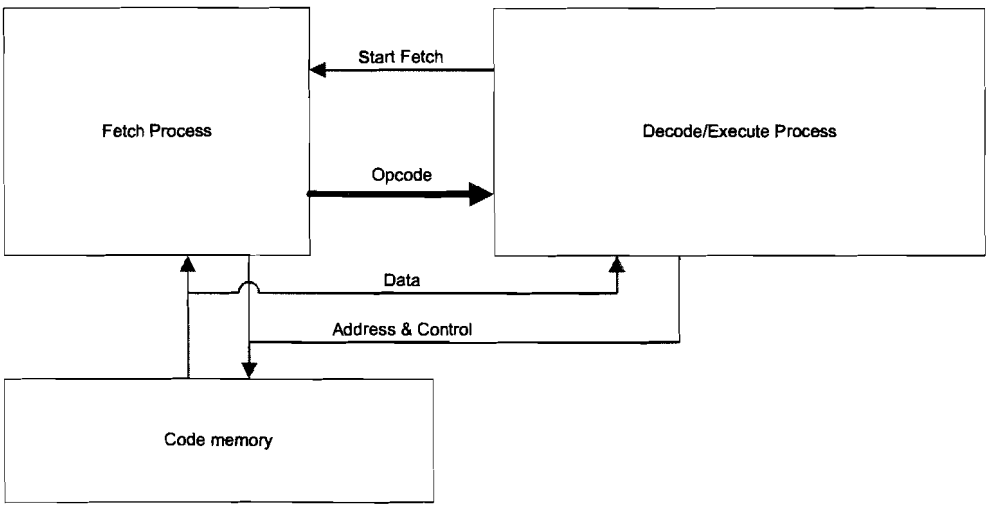
In the HT80C51-SU the instructions are divided into four different categories by the decoder.
The first category contains the regular instructions and allows instructions to perform pre-fetching.
The second category contains the branch instructions with or without computation.
The third category contains instructions which deal with the external memory.
The fourth and last category handles special ALU instructions as the multiply and divide instructions.

These categories are each divided into several sub categories. Only the first main category can perform a pre-fetch of a new instruction while it is executing its current instruction. The other categories have shared resources with the fetch unit and will first finish the execution of the instruction before fetching a new instruction. The pre-fetch unit is not an autonomous process, it only starts fetching upon request and will be a regular fetch after the completion of an instruction in case of the three latter categories. The global architecture of the initial HT80C51-SU is shown in [Figure 13].



[Figure 13] HT80C51-SU data flow chart

The HT80C51-SU consists of two processes, operating in parallel. The first is the fetch process, which can fetch the instruction opcode after a request of the second process, the decode/execute process.



[Figure 14] HT80C51-SU communication between Fetch and Decode/Execute

The address (PC) for fetching the opcode from the code memory is calculated in the execute process. Only here it is guaranteed that the PC and the code memory have no other dependencies. When the PC is calculated and there are no more dependencies, the execute phase will communicate this to the fetch process through a dedicated channel. This channel (*Start Fetch* in [Figure 14]) is the wake up call for the fetch process. Normally this is done after the completion of the instruction execution, although the execution process can request the pre-fetch process to fetch the next opcode in some of the categorized instructions. The fetch process itself will ultimately return the opcode to the execute phase through a dedicated channel ("opcode" in [Figure 14]). Thanks to these two dedicated channels the fetch and the execute processes are synchronized in a correct and intuitive manner. The categorization of the instructions is done by the decoder. The opcode is decoded into proper condition signals (similar to the Execution Bits (EB) in the low cost HT80C51 design, refer to 3.3). Each of the four categories sequentially executes a set of conditional sub categories, like the HT80C51-LC.

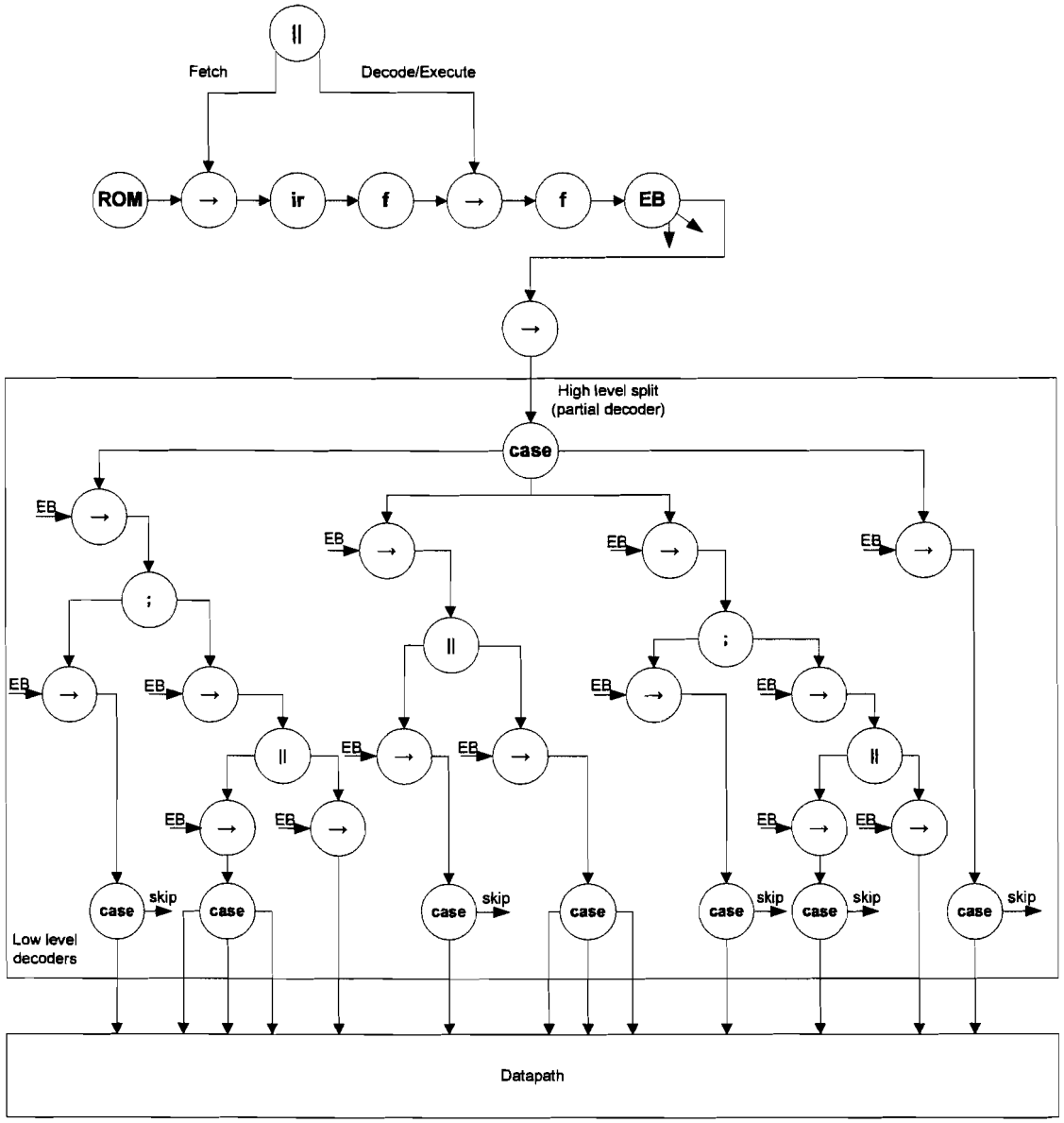
4.2. Datapath

The 80C51 instruction code is analyzed to find an efficient overlap in the datapath, so that logic and communication paths can be shared where possible without suffering in performance. The use of glue logic is avoided whenever possible and data is copied as little as possible in the processes to prevent the increase of complexity of the datapath and control structure. Although the datapath of the HT80C51-SU implementation is comparable with the low cost HT80C51, there are some small differences. Because of the control structure differences the datapath has to be adapted. The data has to be latched at more places, e.g. between the pre-fetch process and the decode/execute processes.

4.3. Control structure

The HT80C51-SU implementation of the control structure is designed to partially split the control flow into four categories. The problem with the HT80C51-LC design is that every case for execution should be evaluated before the next step could be taken. This design decision prevents glue logic because there are no additional invocations to a common or shared resource. The strength of the HT80C51-LC design is that the control structure is very transparent. This saves area, but it prevents speeding up the design. With the HT80C51-SU an initial high-speed step is taken to split up the control flow by a high level decoder. This is necessary when adding pre-fetching to the design.

Only one category allows starting up the pre-fetch unit while it is still executing its instruction, as shown in [Figure 13]. Therefore this category contains as many instructions as possible to exploit the benefits of pre-fetching. The tasks which remain are decoded in the category itself like it is decoded in the HT80C51-LC. This prevents glue logic and therefore the control logic is kept more transparent. The HT80C51-SU design is constructed such that, after the instruction fetch, the control flow partially depends on the actual instruction. The control structure is organized as shown in [Figure 15]. The high level decoder splits up the control flow which makes the control path smaller and different for each instruction group, therefore dependent on the actual instruction.



[Figure 15] HT80C51-SU handshake circuit

The decoder will assign the proper execution signals, the EB. These EB are computed in one large assignment. The decoder will split up instructions in four categories and assign execution signals (EB) such that the proper category and the proper tasks in the categories are being executed.

```
Decode =
[
  << EB0, EB1, EB2, ..., EBX >> := << EBCat0, EBCat1, EBCat2, ..., EBCat3mul >>
]
```

[Code Fragment 12] The Execution bits decoded by the decoder

After the decode phase it is clear which path must be followed by the control structure, to execute the correct tasks of the instructions. The right category and the proper tasks in this category are chosen after the decode phase. Every category consists of a sequence of conditional tasks, each of these conditional tasks contains more sequential and parallel tasks.

```
EB_Category0 =
[
  If EB_Cat0_Read    then doRead0()    fi ;
  If EB_Cat0_Execute then doExecute0() fi ;
  If EB_Cat0_Write   then doWrite0()   fi
]
```

[Code Fragment 13] The Execution bits used in category 0

The same procedure holds for the other categories. Category 1 is for branch instructions. It evaluates the arguments and calculates or assigns the branch address. It is decided that this category will not allow pre-fetching, since the pre-fetch results may have to be discarded as the PC can be changed during execution of the instruction.

```
EB_Category1 =
[
  If EB_Cat1_Read    then doRead1()    fi ;
  If EB_Cat1_Execute then doExecute1() fi ;
  If EB_Cat1_Jump    then doJump()     fi
]
```

[Code Fragment 14] The Execution bits used in category 1

Category 2 is for the special MOV instructions as the *MOVC* and the *MOVX* instructions. These are special *MOV* instructions which move bytes from the external data memory or the code memory. As this category also has access to the code memory it is prevented to initiate pre-fetching, which would lead to memory access conflicts.

```
EB_category2 =
[
  If EB_Cat2_Read    then doRead2()    fi ;
  If EB_Cat2_Execute then doExecute2() fi ;
  IF EB_Cat2_Write   then doWrite2()   fi
]
```

[Code Fragment 15] The Execution bits used in category 2

The last category 3 executes only the *DIVIDE*, *MULTIPLY* and *NOP* instructions. These instructions are relatively transparent for the control structure as can been seen in [Code Fragment 16]. Only one *IF* construct is used to execute the proper logic. Pre-fetching is possible with these instructions, but it is not implemented. The control overhead for implementing another category would lead to a slower average circuit while the gain in performance is small for these rarely used instructions.

```
EB_category3 =
[
  If EB_Cat3_Div      then doDivide()
  or EB_Cat3_Mul      then doMultiply()
  else                     skip
  fi
]
```

[Code Fragment 16] The Execution Bits used in category 3

4.4. Results

The HT80C51-SU is designed in Haste and completely compiled with the Handshake Technology design flow into a standard-cell CMOS 0.14 µm netlist. The same testbench, previously used for benchmarking the low cost HT80C51-LC in section 3.5, is used to analyse the HT80C51-SU. Again to determine the execution speed we take the average time it takes to run the benchmark program. The total amount of time it takes to execute the testbench on the HT80C51-SU is 12800 ns. This is a substantial gain in comparison with the HT80C51-LC described in 3.3 which takes 17800 ns. On average it is 40% faster.

Instr. Group	ROM fetch	RAM read	RAM write	Total instr time	ROM time + OH	RAM time + OH	Cate-gory	Profit prefetch time
	#	#	#	ns	%	%	#	ns
1	1	0	0	42-128	16-50	0	0 – 3	19
2	1	0	1	71-87	24-30	24-30	0 – 2	20
3	1	1	0	77-88	24-27	24-27	0 – 2	19
4	1	1	1	85-90	23-25	46-50	0 – 2	20
5	1	2	0	85-106	20-25	39-50	0 – 2	19
6	1	2	1	110	19	57	0	20
7	1	3	0	115	18	55	1	-
8	2	0	0	74-100	42-56	0	0 – 1	19
9	2	0	1	86	49	24	0	20
10	2	0	2	121	35	35	1	-
11	2	1	0	94-108	39-45	19-22	0	19
12	2	1	1	86-112	38-49	18-24	0	20
13	3	0	0	107-118	53-59	0	0 – 1	-
14	3	0	1	119	53	18	0	-
15	3	0	2	122	52	34	1	-
16	3	1	0	110	57	19	0	-
17	3	1	1	106-113	56-59	18-20	0	20
18	3	2	0	112	56	37	0	-

Note: OH: Overhead

[Table 16] HT80C51-SU memory usage

Regardless of the fact that the HT80C51-SU implementation uses a pre-fetch unit, the fetch and decode phase did not changed much. The main transformation is in the control structure. After the opcode fetch the control path is split up into one of the four main categories. Each of these categories then follows a fixed control path with executions of a part of the instruction or skipping the execution of the other tasks in the sequence of tasks the category contains.

The increase in performance is largely due to better control structure in the execute phase and that a lot of instructions support pre-fetching. The major part of the 160 different instructions, 113, uses the possibility of pre-fetching. For these instructions the increase in performance can go up to 20 ns per instruction (profit in time with pre-fetch, seen in [Table 16]), that is a profit of 15% up to 48% for some instructions. The other instructions have shared resources between the fetch phase and the execute phase and follow the traditional execution scheme. The total gain of pre-fetching alone is 20% in this testbench.

The increase in performance due the better realization of the control structure is 19%. The instructions do not need to follow the complete execution scheme and execute or skip every task like in the HT80C51-LC.

There is still room for a major improvement in terms of performance, but this requires a more radical approach. Still the main part of an instruction is fetching data from the code or data memory, see [Table 16] column *memory time with overhead*. In the HT80C51-SU design a memory access takes also a complete 21 ns to complete. Some data memory read accesses can be done in parallel with the argument read accesses from the code memory, but five memory accesses will take in general 105 ns to complete. See for example instruction group 15 in [Table 16]. In which 86% of the total instruction time is spent on memory accesses! There are possibilities to reduce the amount of memory accesses and to speed up the entire instruction scheme. This is treated more specifically in chapter 5.

5. HT80C51-HS

The speed up design of the HT80C51-SU, as introduced in chapter 4, is a truly better design with respect to speed than the HT80C51-LC, the starting point. Nevertheless the HT80C51-SU is still lagging improvement in terms of speed as targeted in the problem description (section 1.1). Therefore a more rigorous approach has to be taken. For this we have identified the well-know principle of pipelining, refer to [1].

5.1. Conceptual architecture

The two main problems in the previous HT80C51 designs are:

1. The logic adds much control delay which increases throughput time;
2. There are many sequential memory accesses (both code and data memory) for an instruction.

The solution for the first problem is to increase the instruction throughput by pipelining the HT80C51 architecture (HT80C51-HS, High Speed). To eliminate the second problem we propose two solutions. First a central fetching unit which fetches 32 bits instead of the traditional 8 bits for the code memory (refer [23]). This reduces the number of accesses to the code memory. Second, declare the frequently used RAM space as internal registers. So besides the DPTR, SP, Accumulator and B registers, the R0..R7 become internal registers, which are directly accessible. This way there is no need to address the data memory for accessing these frequently used registers. R0..R7 are accessible via two ways. First approach is through a dedicated bus when calling an instruction like *ADD R0*. Second approach is that the decoder will check the address when the data memory will be accessed and reroute this address to the registers when R0..R7 is supposed to be accessed, for example the *ADD address* instruction where the *address* is the R0..R7 address.

5.2. Pipelining and memory architecture

When going for a pipelined design of a microcontroller many issues have to be taken into consideration. It is not only important to identify the most balanced way of pipelining, but also problems like data dependencies between different stages of the pipeline have to be taken into account.
Pipelining is an implementation technique where instructions can be executed simultaneously. As seen before, an instruction consists of several parts of execution. When splitting these execution parts into independent tasks by dedicated hardware, it is possible to execute multiple instructions at the same time. The individual times of executing a single instruction will most likely increase (latency), because of the increased complexity of the control structure and the increased length of the datapath (extra registers). However, due to the simultaneous execution of multiple instructions in parallel the total throughput of instructions increases. The increase in instruction throughput means that a program run on the processor executes faster and has a lower total execution time, even when no single instruction executes faster on its own.

As multiple instructions are being executed simultaneously, it is possible that instructions in the pipeline depend on each other. These situations are called *hazards* and they will prevent the next instruction from being executed while other instructions are still in the pipeline. Hazards reduce the performance from the ideal speedup gained by pipelining and therefore should be prevented as much as possible. There are three types of hazards:

- The first is *Structural hazards*, these happen for example when separate parts of the pipeline want to use one shared resource. E.g. the fetch phase fetches the opcode from the code memory and the execute phase needs to access the code memory for its operands. Consider the next code sequence:

```
MOV dir, dir
ADD A
```

[Code Fragment 17] Example of a structural hazard

The MOV instruction is still reading its arguments from the code memory when the first stage wants to fetch the next instruction from code memory.

The 80C51 architecture can exhibit different structural hazards that we need to address. The following shared parts of hardware have been found and need to be addressed.

1. Sharing of the code memory
2. Sharing of the data memory
3. Sharing of the extended data memory
4. Sharing of the SFR memory
5. Sharing of the ALU

- The second type of hazards is *Control hazards*. Control hazards, also called branching hazards, occur when the controller is told to branch. E.g. when an instruction modifies the code memory address (PC), while the next instruction is already being fetched.
  Consider the next code sequence:

```
   JMP  label
   MOV  dir, A

label:
   INC  A
```

[Code Fragment 18] Example of a control hazard

The JMP instruction will cause a control hazard because the PC is changed while the next instruction is already being read from the code memory and possible partially being executed.

The 80C51 architecture can exhibit different control hazards that we need to address. The following different types of branches are been found in the 80C51 instruction set.

6. Conditional branches
7. Unconditional branches

- The third and last type of hazards is *Data hazards*. Data hazards are created when an instruction depends on the result of a previous instruction which is still being executed.
  Consider the next code sequence:

```
MOV  A, #data
INC  A
ADD  A, #data
ORL  A, #data
```

[Code Fragment 19] Example of a data hazard

The MOV instruction will write back data to the Accumulator, but it is possible that the INC instruction already read the Accumulator before the write of the MOV instruction is finished.

The 80C51 architecture can exhibit three different types of data hazards

- Read After Write (RAW), a RAW hazard exists if an instruction wants to read a data object but the previous instruction still has to write the same data object. In [Code Fragment 19] this happens between the move and the increment instruction. If the increment instruction read the accumulator before the move instruction has written the accumulator, it will be incrementing the wrong value.
- Write After Read (WAR), a WAR hazard exists if an instruction wants to write a data object before the previous instruction has read to the same data object. In [Code Fragment 19] this is the case between the increment and the add instruction. If the add instruction writes the value to the accumulator before the increment instruction is able to read, it will also be incrementing the wrong value.
- Write After Write (WAW), a WAW hazard exists if two instructions write to the same data object in the wrong order. In [Code Fragment 19] this can be the case between the increment and the addition instruction. If the increment instruction writes later to

the accumulator than the addition instruction, the memory location will contain the incorrect value afterwards.

There is also a Read After Read (RAR) dependency but this never causes a hazard.

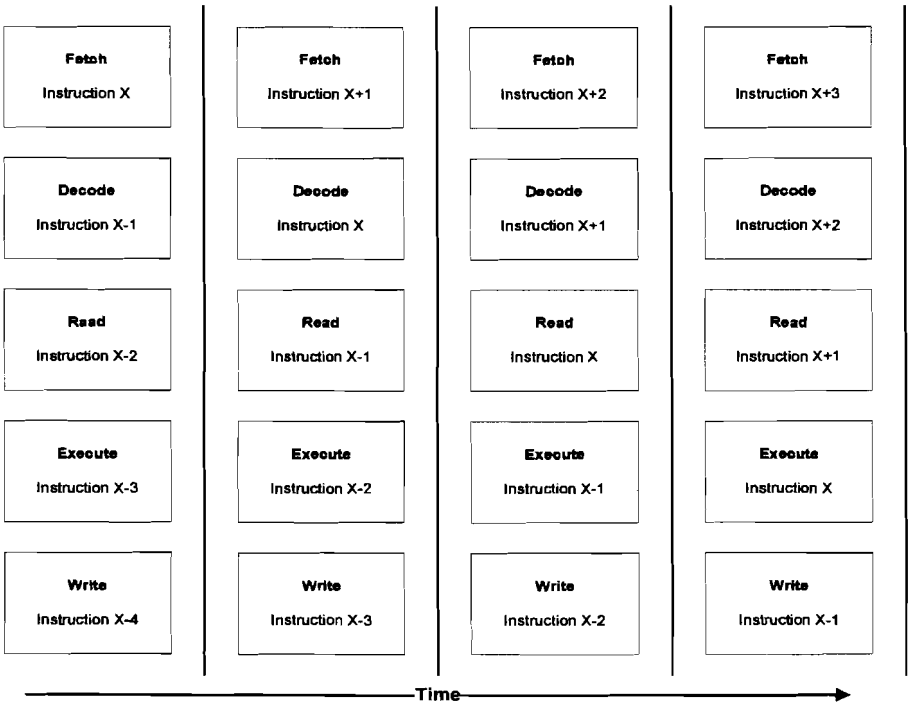
The 80C51 architecture can exhibit three different types of data hazards that we need to address. In the 80C51 instruction set the following data objects are found which can introduce data hazards.

8. Accumulator
9. B-register
10. R0..R7
11. DPTR
12. PC
13. PSW-register (ACC flags)
14. SP
15. Memory location

All these hazards should be tackled in a systematic way. Hazards can be eliminated in different ways, in a dynamic or in a static way. As we want to use existing code compilers and existing code the static elimination is not an option. The dynamic solution is to find an architectural solution to every possible hazard. Throughout the remainder of chapter 5, every possible hazard is addressed.

In order to get the maximum performance out of the pipeline architecture while maintaining correct behaviour, mechanisms are needed to control the data flow, accesses to the memory and the use of shared hardware. Structures as data forwarding and local communications could be used to solve these dependencies without suffering in performance.

Structural hazards like dependencies to the code memory (refer to dependency 1 on page 44) are eliminated by having only one place to access the code memory. This is done by the second architecture choice: fetching a complete word (32 bits) access instead of a byte.



[Figure 16] HT80C51-HS pipelined operation

Only the fetch phase will use the code memory such that dependencies (structural hazards) do not exist in case of code memory. When there are branches, some of the memory accesses can be redundant because the pipeline can be filled with unwanted instructions; this is a disadvantage for power consumption. The data fetched from the code memory contains both the opcode of the instructions and the arguments for some of the instructions.

Other shared memory resources as the frequently used Rn registers which are normally addressed in the data memory range can be declared as internal registers. This will speed up the design significantly, as the registers are accessible in a direct way instead via an addressable bus, but decoding is still needed. There are much less code memory accesses (on average less than 1 for each instruction) due to using a 32-bit memory accesses, and with the Rn registers in the internal registers the amount of data memory accesses will be greatly reduced. [Table 12] and the table in appendix A5 will be changed dramatically as the instruction groups will change. [Table 17] shows the new partition of instruction groups for the HT80C51-HS. The structural hazards with data dependency for these registers and data memories remains, but there are effective solutions for this problem. These are discussed in the following sections.

5.3. Choice of pipeline structure

The HT80C51-HS (High Speed) will be a fully pipelined architecture with a 32 bits wide access to the code memory. This 32 bits access to the code memory eliminates the need to fetch operands from the code memory after the decoding operation of the instruction. The 32 bits access to the code memory also simplifies the pipelined architecture and reduces hazards (e.g. structural hazards) and stalls in the pipeline which can happen otherwise.

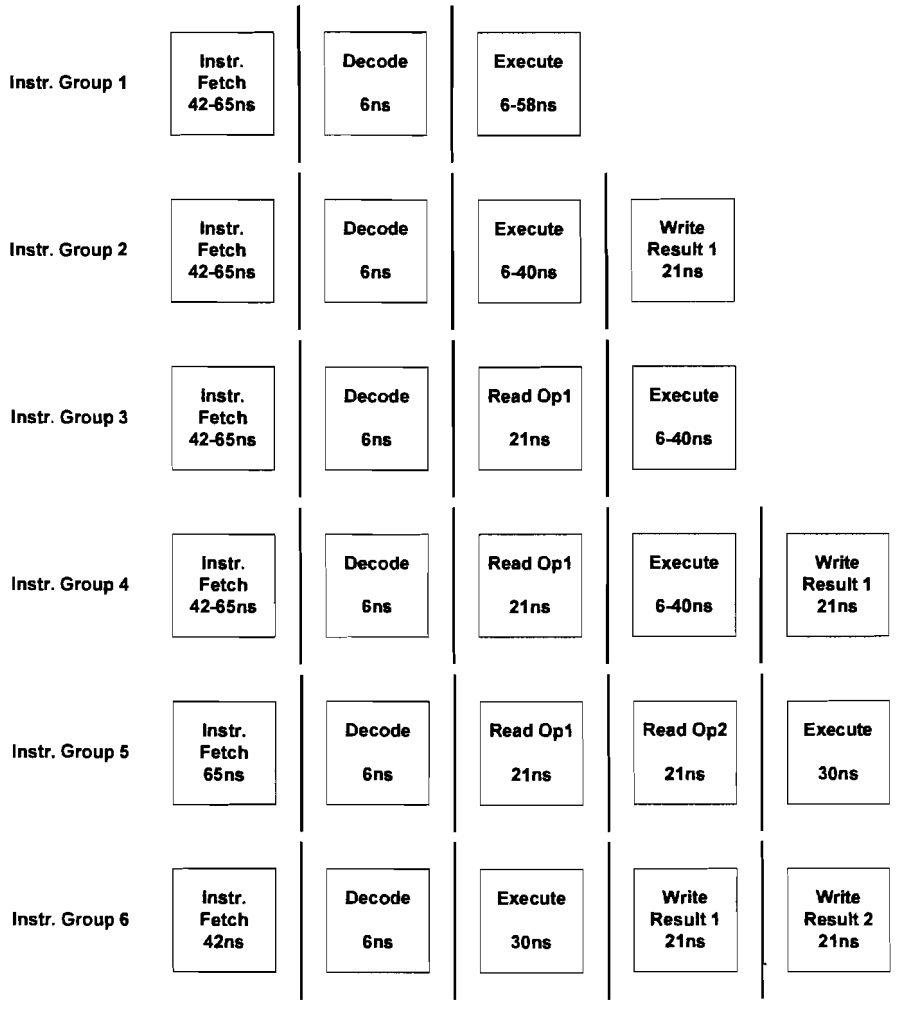
The instruction scheme will change significantly compared to the HT80C51-SU due to the 32 bits fetch unit. When residing the R0..R7 (Rn) register locally in variables instead of in the data memory address space the result of instruction regrouping is [Table 17].
There are only 6 instruction groups left in comparison to the 18 instruction groups of the HT80C51-SU [Table 16]. These instruction groups are now divided by the amount of data memory accesses (RAM Read and RAM Write).
There are only a maximum of two RAM reads or two RAM writes memory accesses compared to the three of HT80C51-SU. And there are much fewer memory accesses than before, only 103 instructions of the 255 (40%) need a data memory access instead of 203 (80%) of the HT80C51-SU.

Instruction group	RAM Read	RAM Write	Total instructions	Example of instructions
#	#	#	#	
1	0	0	152	NOP, Function A, Rn (Un)conditional branches
2	0	1	18	Unconditional branches (CALL)
3	1	0	46	Function dir, A (dir, Rn) Conditional branches
4	1	1	28	Function dir, dir PUSH, POP
5	2	0	2	Unconditional branches (RET)
6	0	2	9	Unconditional branches (CALL)

[Table 17] HT80C51-HS instruction groups

Each of these six instruction groups take different times and steps to execute. Pipeline structures, based on the instruction regrouping [Table 17], reflect the characteristics of each group, defining a specific stage configuration as shown in [Figure 17].
A typical instruction scheme and pipeline configuration applies to instruction group 4. Instruction group 1, 2 and 3 pass through some unnecessary parts of the instruction scheme (like reading and writing to the data memory for instruction group 1). Instruction group 5 and 6 pass through some unnecessary parts, but also loops multiple times through parts of the instruction scheme.

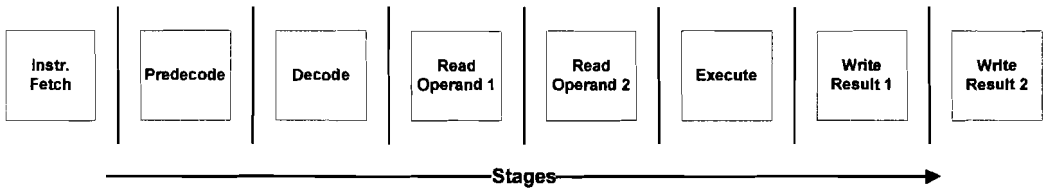
[Figure 17] shows the six instruction groups with the time it takes to execute the tasks of which they exist. These times are derived from the HT80C51-SU microcontroller. All the times, but two, are a precise time. The time of the fetch and the execute task is not fixed and can be variable depending on the type or length of the instruction.



[Figure 17] HT80C51-HS instruction groups scheme

To balance the pipeline in well-tuned stages we need to know which parts will be separate blocks and which processes can be combined. Due to the overhead asynchronous designs give for controlling parallel processes and potential hazards, the need exists to not introduce an architecture with a pipeline which has too many stages and therefore is control heavy.

The first task, Instruction Fetch, is a relatively slow task. This task needs to fetch 32 bits at once from the code memory and calculate complete instructions. Complete instructions will be a single, two or three byte instructions. Experiments show that it is possible to execute this task in about 65 ns for a three byte instruction. It is recommendable to split up this task into two separate tasks to increase the throughput. Therefore this task will be split into a separate fetch task and a separate predecode task. This split will change the complete overview for a pipelined 80C51 microcontroller. [Figure 18] shows a pipelined architecture for the 80C51. This architecture is an extreme variant and all tasks (or stages from now on) should be evaluated if they are necessary to increase the total throughput of the 80C51 microcontroller.



[Figure 18] Extreme pipeline

[Figure 18] shows 8 stages. The first two have already been mentioned; the actual *Instruction Fetch* and the *Predecode* which calculates the complete instruction. The following stage is the actual *Decode* stage which decodes the instruction into executable tasks. The following two stages are the *Read Operand* stages. These will gather the operands from the data memory if needed. The next stage is the *Execute* stage. Everything gathered and decoded by previous stages is being executed in this stage. The last two stages are the *Write Result* stages. The *Write Result* stages will write the calculated result back to the data memory.

It is difficult to determine how often and in which order instructions occur in a normal program code, because every program code is different in operation. As it remains unknown which code will be run on the microcontroller it is not possible to determine the most optimal architecture.

[Table 17] and [Figure 17] show that there are a few instructions with double read or write accesses to the code memory.
Only 9 instructions of the 255 have double write accesses to the data memory. All of the double write backs are *CALL* instructions. These write backs do not need to be calculated, so they do not need any execution (ALU) time. Only an internal register (PC) needs to be transferred to the memory interface. There is no need for complicated calculations. Complicated calculations like the *MUL/DIV* instructions will write back the calculated data to the fast internal registers A and B.
Only 2 of the 255 opcodes are double read accesses to the data memory. We conclude that the need to introduce two separate pipeline stages for these 4% of total instructions is not very high.

There are 55 (28+18+9) instructions with write accesses to the data memory. This is 22% of the total amount of instructions. There are 76 (28+46+2) instructions with read accesses to the data memory. This is 30% of the total amount of instructions. If there will be separate stages for the read and write accesses there will be the need for extra control logic (aside from the extra registers and channels between the stages) to arbitrate and schedule the read and write accesses between mutual instructions.

The decision is made to go for a four stage pipeline. This does not imply this is the best solution to execute a given program the fastest as possible, but this is the trade off made.
The first stage (IF) will decouple the pipeline stages from the code memory. The task of this stage is to fill the internal buffers with code memory fetches and handle the interrupts correctly. This stage will forward the fetched data through a dedicated handshake channel to the second stage, the predecode stage which gather the complete instruction. The third stage will decode the instruction completely and handle all the necessary reads of operands. This is mostly done from the already available fetched operands or the internal registers. Otherwise (30%) the read is done to the slower data memory. Since

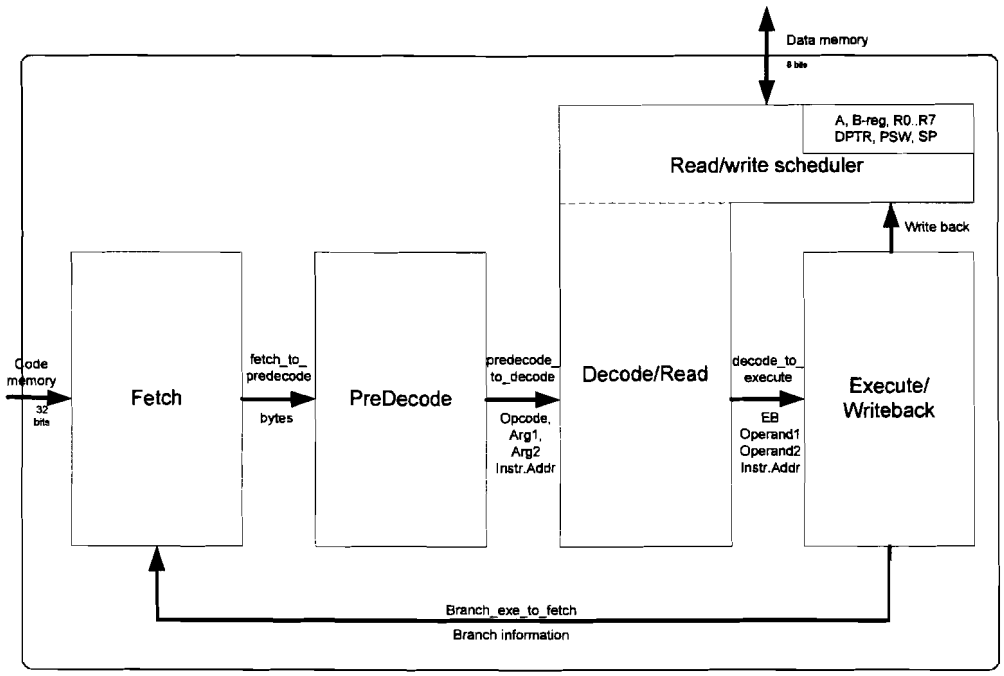
the actual decoding of an instruction is done faster than the fetching or executing of the data, the access to these internal registers and data memory is also done in the decode stage. This way the pipeline is more balanced

The fourth and last stage will be the execution stage which also will write back the calculated data. Only 22% of the instructions will write back the calculated result to the slow data memory. The other instructions use the fast internal registers. So the main part of the data written back can be done in 6 ns (time to assignment to an internal variable) instead of 21 ns (single data memory access) or 42 ns (double data memory access) needed for a write back to a data memory.

When there is no conflict between the read and write process it is possible to execute these stages simultaneously without any stalls in the pipeline. In the last section of this chapter you can see an example of this parallel behaviour. Two wave forms, one from an execution of an instruction with a parallel read and write [Figure 32]. The other waveform shows a conflict between the two processes and will first finish the write back of the result by stalling the read phase of the other instruction [Figure 33].

5.4. Detailed architecture

The HT80C51-HS architecture will be divided into four pipeline stages: fetch stage, predecode stage, decode stage and execute stage. A simplified diagram of the operation of a pipeline is presented in [Figure 16]. The fetch stage addresses the ROM and collects 4 bytes (word) at once from the code memory and communicates it to the predecoder which will form a complete instruction. The complete instruction with its arguments is then passed through to the decode stage. The decode stage decodes the instruction into different Execution Bits (EB) and will gather all the operands needed for the execution of the instruction by reading the data memory or the internal registers. The decode stage will also calculate the destination addresses for the result of the execute stage. The execute stage processes all the information gathered by the previous stages, executes the instruction and writes back the result to the address calculated by decode stage. The global architecture of the HT80C51-HS is shown in [Figure 19].



[Figure 19] HT80C51-HS global architecture

[Code Fragment 20] presents the top-level pseudo code. There is no feedback from the decode stage and execute stage to the fetch stage except on branches or interrupts for which it is possible to write the new branch conditions to the fetch stage. All stages remain operating normally in case of a branch. The decoder stage marks an instruction as a branch, when this occurs the execute stage will calculate the branch address and perform a channel communication to the fetch stage with the newly calculated address if needed (only when the branch is taken). The branch_exe_to_fetch channel which indicates the branch is sampled and buffered in the fetch stage to prevent deadlock. The next fetch to the code memory will then be with the code address previously calculated by the execute phase. The new data fetched by the fetch stage will be marked as a branch packet. The data processed by the fetch, predecode, decode and execute stage between the initial decode of the branch and actual new execution of the new instruction on the branch address is flushed and not used in the execute stage. All data written back from the execute stage to the decoder stage is discarded, so the registers and RAM data are remaining valid.

A more detailed explanation of the HT80C51-HS is given in the following sections.

```
cpu.ht =
[
  forever do
    fetch     ( fetch_to_predecode!
              & branch_exe_to_fetch?
              )
  od
||
  forever do
    predecode( fetch_to_predecode?
              & predecode_to_decode!
              )
  od
||
  forever do
    decode    ( predecode_to_decode?
              & decode_to_execute!
              & write_back?
              )
  od
||
  forever do
    execute   ( decode_to_execute?
              & branch_exe_to_fetch!
              & write_back!
              )
  od
]
```

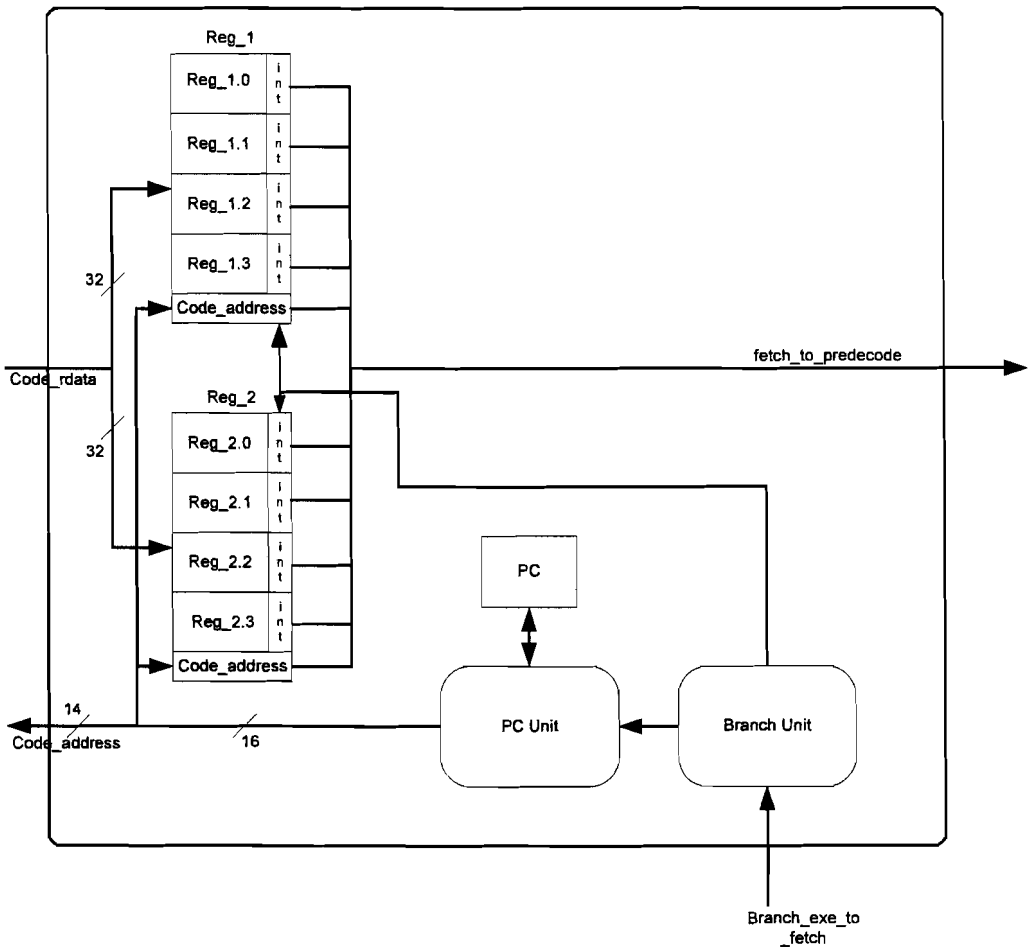
[Code Fragment 20] HT80C51-HS The execution of the execute stage

5.4.1. Fetch stage

The fetch stage of the pipelined HT80C51-HS follows a completely different approach compared to previous designs. It is not a remote procedure call like the previously described pre-fetch process of the HT80C51-SU in chapter 4, but a completely autonomous stage.

The most important design decision was to allow a complete word (4 bytes) to be fetched each time from the instruction memory instead of a single byte. This scheme reduces the average instruction memory overhead by a factor four, since only once an address needs to be calculated for a ROM access which fetches a complete word instead of a single byte.
Another beneficial point of this approach is the reduction of control hazards in this part of the design, since the fetch stage is the only place which accesses the code memory (refer to dependency 1 on

page 44). The execute stage will not have to address the instruction memory and gather the operands needed for an instruction.



[Figure 20] HT80C51-HS fetch stage block diagram

The fetch stage consists of two 4 bytes registers (Reg_1 and Reg_2) to store two complete words that are fetched. The two registers are used as wagging buffers. This way the flow of instructions to the decode stage is never interrupted. The principle of wagging buffers will be explained later in this section.

There is no branch prediction in the HT80C51-HS, but it can be implemented in the fetch stage. [Figure 20] shows the block diagram of the fetch stage of the HT80C51-HS.

[Code Fragment 21] presents in pseudo code the implementation of the fetch stage. The main task of the fetch stage consists of a process to fill and empty the two 32-bits buffers. The two buffers are set up in a wagging way (refer [12]). When one of the buffers is written, the other buffer can simultaneously be read from. The two buffers are read and written through handshake channels. After this the processes are reversed; the other buffer is then filled while the full one is emptied. The filling of the buffer is done by a process which accesses the code memory.

```
fetch =
[
  (   load_reg_1() || empty_reg_2()
    ; load_reg_2() || empty_reg_1()
  )
  ||
  buf_branch_channel()
]

buf_branch_channel =
[
    branch_exe_to_fetch?buf
  ; internal_branch!buf
]

load_reg_1 =
[
    set_codemem_addr_access()
  ; Reg_1:=code_rdata
  ; calculate_next_pc()
]

load_reg_2 =
[
    set_codemem_addr_access()
  ; Reg_2:=code_rdata
  ; calculate_next_pc()
]

empty_reg_1 =
[
    fetch_to_predecode!<<reg_1.0,instructionaddress,branch>>
  ; fetch_to_predecode!<<reg_1.1,instructionaddress,branch>>
  ; fetch_to_predecode!<<reg_1.2,instructionaddress,branch>>
  ; fetch_to_predecode!<<reg_1.3,instructionaddress,branch>>
]

empty_reg_2 =
[
    fetch_to_predecode!<<reg_2.0,instructionaddress,branch>>
  ; fetch_to_predecode!<<reg_2.1,instructionaddress,branch>>
  ; fetch_to_predecode!<<reg_2.2,instructionaddress,branch>>
  ; fetch_to_predecode!<<reg_2.3,instructionaddress,branch>>
]

calculate_next_pc =
[
    if sample(internal_branch) then
      internal_branch?PC
    else
      PC:=PC+4
    fi
]
```

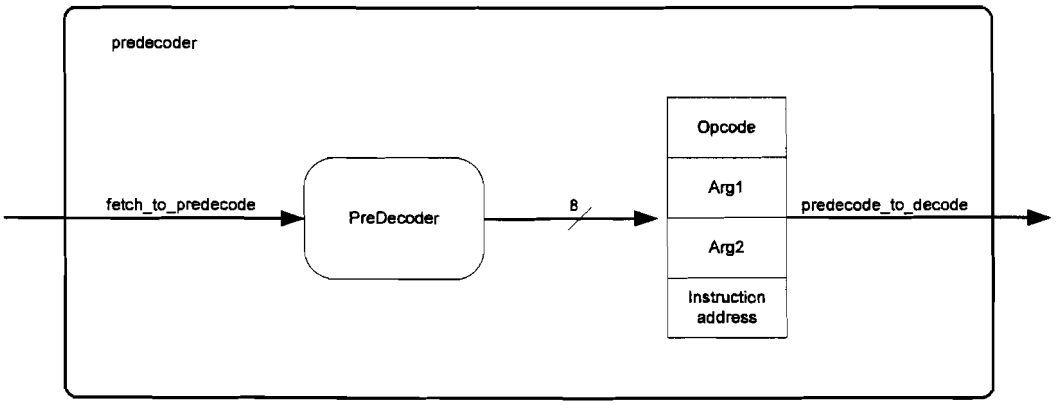
[Code Fragment 21] HT80C51-HS fetch stage pseudo code

First it will setup the address and control and will initiate the handshake to the memory. After this the data is ready to be read from the code memory. When this is finished the PC is adapted by the PC Unit. It is possible to execute this in parallel with the reading of data from the code memory, but this is not done at the moment because it is not time critical. The PC Unit will supply the correct address for the code memory.
The PC Unit will be driven from the Branch Unit, and can change the PC (i.e. whether or not the previous instruction is a branch, jump or MOVC. Refer to dependency 12 on page 45) in case of a branch taken. Normally the 16-bit PC is incremented by four during every read cycle of the code memory by the fetch stage (because of a complete word fetch, four bytes). In case of a branch the *buf_branch* procedure will decouple the execute stage to prevent deadlock in the execute stage. A sample is needed to safely detect that the branch buffer is filled. When a code address is taken which is the branch address the branch flag in Reg_1 or Reg_2 is such that the predecode stage can correctly find the start of the instruction fetched form the branch target address.

The address incrementing takes place independent of whether the next sequential address is needed or not (speculatively). The PC Unit uses its own adder to increment the address. This prevents sharing of the ALU adder and it is an advantage regarding speed while it also avoids a structural hazard (refer to dependency 5 on page 44). The disadvantage is that it cost more area. Some instructions (branches and some moves) need the PC in the decode stage or execute stage; the PC is stored together with the Reg_1 and Reg_2 so it can be sent with the instruction to the decode stage and execute stage. Instructions which need the PC in the decode stage or execute stage are for example the *SJMP REL*. When the last byte out of Reg_1 or Reg_2 is sent to the predecoder the register will be filled with a new word from the code memory.
The fetch stage can be seen as a parallel to serial converter. Four bytes enter the converter and will be sequentially transferred to the predecoder. This stage therefore runs at a lower rate than the predecoder.

5.4.2. Predecode stage

Because the 80C51 instructions have variable length (one, two or three bytes), there is a need for a predecode step to send the opcode with the correct amount of arguments to the decode stage. This is done by the predecode stage. The predecode stage collects its data from the fetch stage via a dedicated channel called *fetch_to_predecode* as show in [Figure 21].



[Figure 21] HT80C51-HS predecode stage block diagram

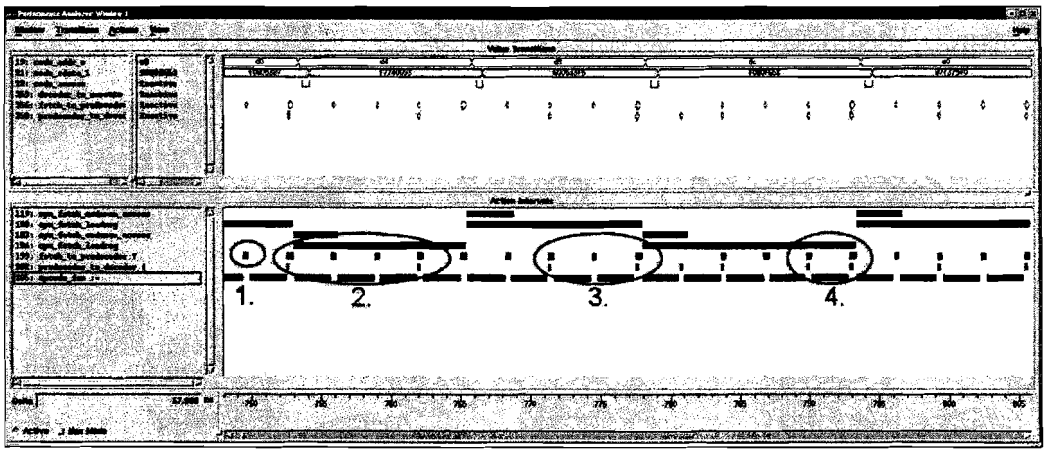
The predecode stage gets the first byte which is the opcode and calculates how long the complete instruction is. It will store the opcode and receive its arguments (if any) and store them to the correct arguments (Arg1 and Arg2). Together with the bytes being gathered from the fetch stage, the instruction address is sent to the predecoder. The gathered information is sent to the decode stage via a channel called *predecode_to_decode*.

```
predecode =
[
    fetch_to_predecode?<<byte,instructionaddress,branch>>
  ; if branch_guard() + opcode_guard() then
        Opcode:=byte
      || calculate_guards()
    or arg1_guard() then
      Arg1:=byte
    or arg2_guard() then
      Arg2:=byte
    fi
  ; if sent_to_decoder_guard() then
        predecode_to_decode!<<Opcode,Arg1,Arg2,instructionaddress>>
      || calculate_guards()
    fi
]
```

[Code Fragment 22] HT80C51-HS predecode stage pseudo code

The decode stage receives the complete instruction: opcode, if needed its operands and the instruction address. It is therefore possible that the decoder receives a single byte instruction with arguments from the previous instruction, but this is disregarded in the decoder. [Code Fragment 22] shows the pseudo code of the predecode stage.

The predecode stage can be seen as a serial to parallel converter. The bytes will enter one at the time from the fetch stage and will be gathered until a complete instruction is formed.
The rate of the predecode stage is data dependent and will be multi rate. [Figure 22] shows the multi rate behaviour of the HT80C51-HS predecode stage.
Circle 1. shows the input byte from the fetch stage.
Circle 2. shows a three byte instruction being fetched from the fetch stage before the complete instruction will be sent to the decode stage.
Circle 3. shows a two byte instruction being fetched from the fetch stage before the complete instruction will be sent to the decode stage.
Circle 4. shows a single byte instruction being fetched from the fetch stage before the complete instruction will be sent to the decode stage.



[Figure 22] HT80C51-HS multi rate behaviour predecode stage

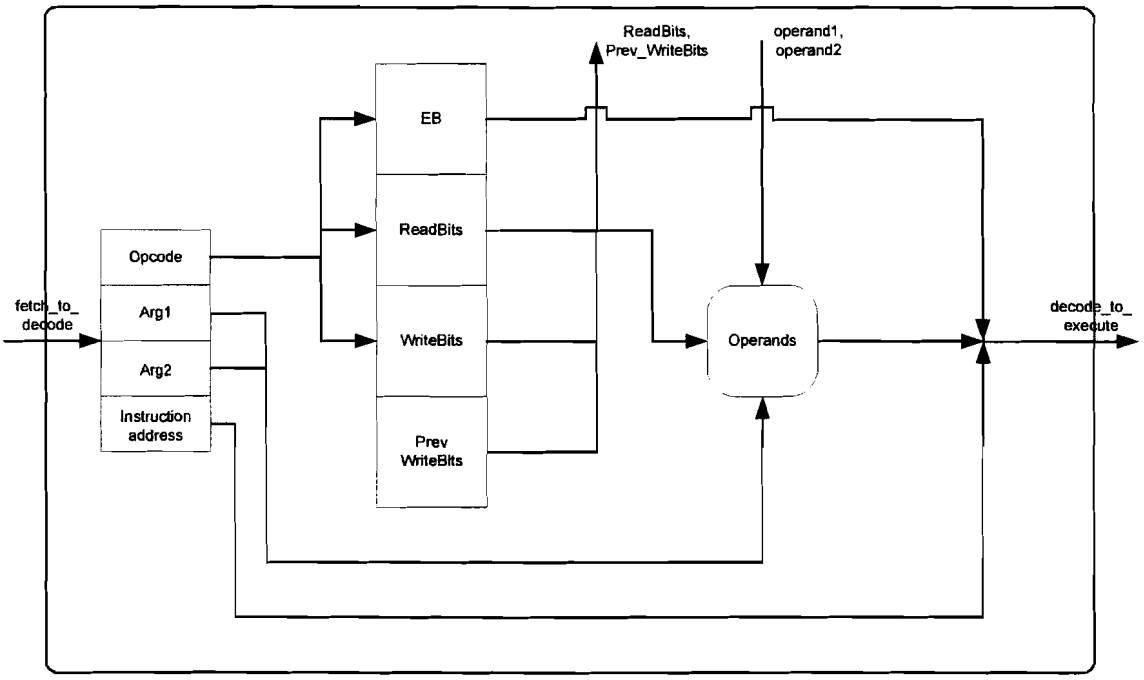
5.4.3. Decode stage

The 80C51 has a complex and irregular instruction set, which makes the decoder complex. The decode stage (see [Figure 23]) always receives the opcode, two arguments and the instruction address (original PC) from the predecode stage. Even in case of a single byte instruction the decoder receives two arguments and the instruction address from the predecode stage. This will keep the control clear and simple as only one channel, with a fixed width between the predecode stage and the decode stage, is used. The decode stage will determine the usage of the arguments. For some instructions there is a need for a source or destination argument but that can be locally decoded from the opcode. The decode stage will decode the opcode (the first byte) of every instruction into a number of Execution Bits (EB). Some instructions need access to the data memory and/or to the local registers (e.g. R0..R7, data pointer, etc). This makes it possible to speed up the access to these commonly used registers as there is no access to the data memory needed. Since the actual decoding of an instruction is done faster than the fetching or executing of the data, the access to these internal registers and data memory is also done in the decode stage. This way the pipeline is more balanced. ReadBits and WriteBits are decoded to identify which data needs to be read from or, after the execution of the instruction, written to.

It is possible that the execute stage will write data back to the destination from the current instruction while also data is being read for the next instruction. This can be handled in parallel by the remote procedure call *read/write scheduler* as long as there is no reading and writing to the same source. If this is the case the read/write scheduler will wait until the data of the current instruction is written before it will read the data (operands) of the next instruction.

For example, when first the instruction *MOV A,#data* and then instruction *INC dir* is being executed, the write back data from the first instruction (MOV) to the Accumulator (A) is being written while the second instruction (INC) will perform a read access to the data memory. This is permitted because there are no data dependencies between the write and read data.

When there are data dependencies between the sources, e.g. the first instruction is a *MOV A,#data* and the next instruction to be executed is *MUL AB* then the decoder needs to wait until the write back data is received from the execute stage via channel *execute_to_decode*. How often these dependencies occur depends fully on the program which is executed by the HT80C51-HS microcontroller.



[Figure 23] HT80C51-HS decode stage block diagram

There are methods as data forwarding that make these dependencies redundant, but they are not implemented (yet) in the HT80C51-HS. Some more research needs to be done to see how effective methods as data forwarding are in the HT80C51-HS architecture.

The next code fragment presents the decode stage in pseudo code. The decoder first receives the complete instruction (opcode and arguments) with the corresponding instruction address.
After this the ExecutionBits, ReadBits and WriteBits are decoded and sent to the remote procedure call *read/write scheduler*. The ExecutionBits are control bits for the execute stage. The ReadBits and WriteBits are the control bits for the read and write processes to the registers and memories. The read/write scheduler will collect the correct operands needed by the instruction in the execution stage. Finally the execution stage will be driven with all the necessary information (Execution Bits, all operands and the instruction address) to execute the instruction.

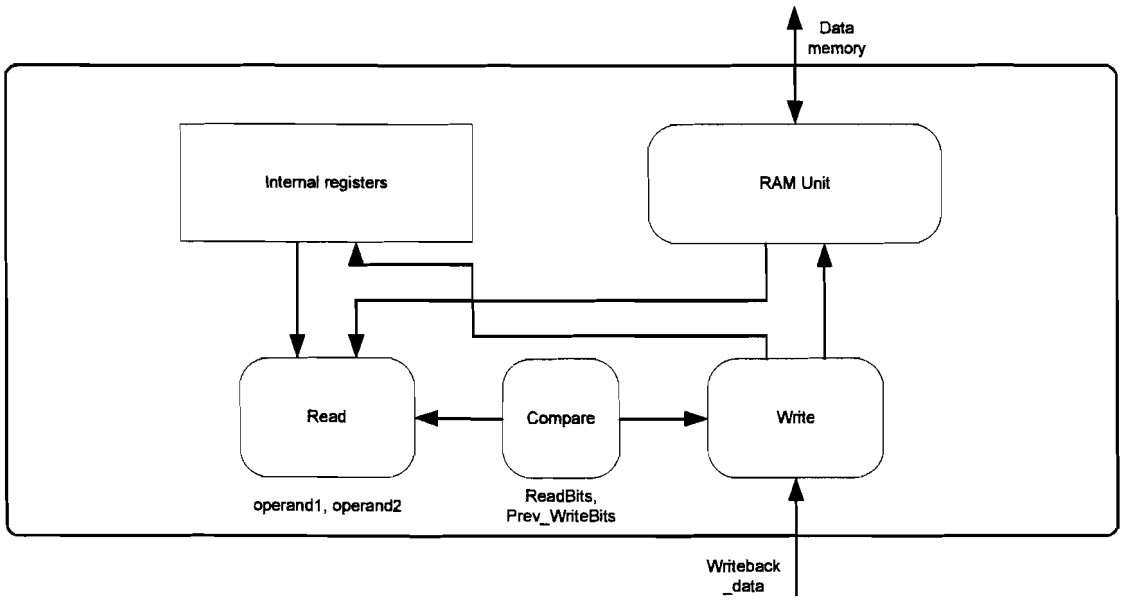
```
Decode =
[
    predecode_to_decode?<< Opcode, Arg1, Arg2, instructionaddress>>
  ; EB(Opcode) || ReadBits(Opcode) || WriteBits(Opcode)
  ; read/write_scheduler(ReadBits, WriteBits, Prev_WriteBits)
  ; decode_to_execute!<<EB, operand1, operand2, instructionaddress>>
  ; Prev_WriteBits(WriteBits)
]
```

[Code Fragment 23] HT80C51-HS decode stage pseudo code

5.4.4. Read/write scheduler

The remote procedure call *read/write scheduler* controls the accesses to the data memory, external data memory, SFR-registers and the internal registers (e.g. R0..R7, data pointer, etc) and is part of the decode stage. On consecutive instructions, structural dependencies (refer to dependency 2, 3 and 4 on page 44) and data dependencies (refer to dependency 8, 9, 10, 11, 13, 14 and 15 on page 45) need to be addressed by the scheduler to guarantee a correct behaviour in a pipelined architecture.

The read/write scheduler will check if the previous WriteBits (the WriteBits from the current execution which is being executed by the execute stage) conflict with the ReadBits of the next instruction. In case of a conflict first a write access to the corresponding destination will be done, when the write access is finished the read access will be initiated and finished. If there is no conflict the read and write process will be done in parallel. When both the write process and the read process are finished the results will be sent to the execute stage and the WriteBits will be copied to the Prev_WriteBits. [Figure 24] shows the block diagram of the read/write scheduler.



[Figure 24] HT80C51-HS read/write scheduler block diagram

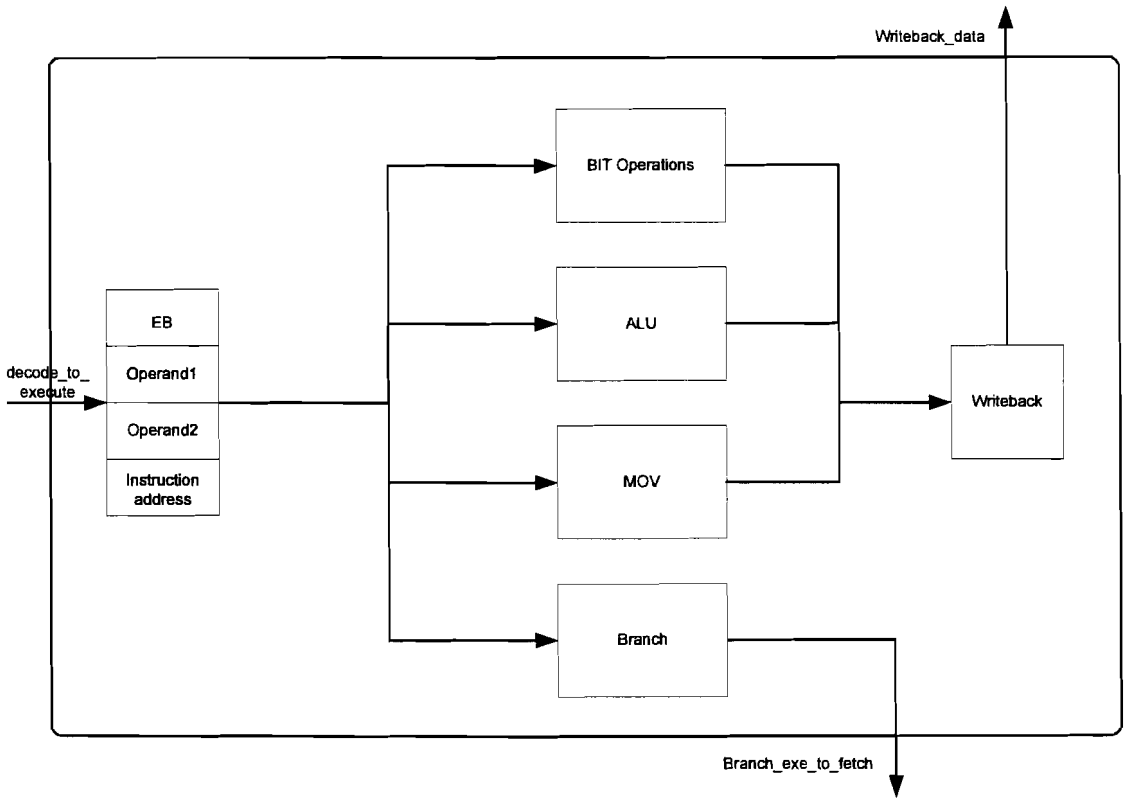
[Code Fragment 24] shows the read/write scheduler pseudo code. When there is a conflict (between the ReadBits and the Prev_WriteBits) the code will first execute the write process and sequential to that the read process. If not, the read and write process will execute simultaneously.

```
read/write_scheduler =
[
  if conflict() then
    write()  ; read()
  else
    write() || read()
  fi
]

write =
[
    writeback_data?<<Writeback_data>>
  ; if Prev_WriteBits.A then
      Write_A(Writeback_data)
    or Prev_WriteBits.RAM
      Write_RAM(Writeback_data)
    or Prev_WriteBits.Rn
      Write_Rn(Writeback_data)
    or …
      …
    fi
]

read =
[
  <<operand1,operand2>>:=
  <<if ReadBits.op1_A then
      Read_A()
    or ReadBits.op1_RAM then
      Read_RAM()
    or ReadBits.op1_Rn then
      Read_RN()
    or
      …
    fi
  , if ReadBits.op2_A then
      Read_A()
    or ReadBits.op2_RAM then
      Read_RAM()
    or ReadBits.op2_Rn then
      Read_RN()
    or
      …
    fi
  >>
]
```

[Code Fragment 24] HT80C51-HS read/write scheduler pseudo code

5.4.5. Execute stage

The execute stage of the HT80C51-HS is set up more or less the same as in the HT80C51-SU design. There is one high level decoder which splits different instruction groups. Because the instructions which need to be executed are delivered complete to the execute stage, with its operands, there is only a need for calculation. The high level decoder splits up the design in different categories e.g. ALU instructions, MOV instructions, BIT instructions and branch instructions. For some instructions the only executing task in the execute stage is moving some data and writing this back to the decode stage. While other tasks are multiplying two bytes and setting or resetting some flags. All the results are sent back to the decoder stage to be stored in the correct destination. [Figure 25] shows the block diagram of the execute stage.



[Figure 25] HT80C51-HS execute stage block diagram

As shown in [Code Fragment 25] the execute stage will first perform a communication with the read/write scheduler to write back data previously calculated by the execute stage. This is done because the write back in the read/write scheduler can be done in parallel with the read process. This needs to be finished before the communication with the decode stage to receive the next execution bits, operands and instruction address. If this sequence is not obtained a deadlock may occur. Finally it will execute the actual execution of the instruction. In case of a branch instruction it is possible that there is a communication to the fetch stage to take the branch. The execute stage will flush every instruction it will receive from then until it receives the instruction from the branch target address previously sent.

```
Execute =
[
    writeback_data!write_data
  ; decode_to_execute?<<EB, operand1, operand2, instructionaddress>>
  ; do_execute()
]

do_execute =
[
    if flush then
      skip
    else
      if EB.bit then
        write_data:=function_bit(operand1)
      or EB.ALU then
        write_data:=function_alu(operand1, operand2)
      or EB.mov then
        write_data:=function_mov(operand1, operand2)
      or EB.branch then
        execute_to_fetch!<<function_branch(instructionaddress, operand1)>>
      fi
    fi
]
```

[Code Fragment 25] HT80C51-HS execute stage pseudo code

Branch Operations
A branch operation will calculate a new instruction address for accesses to the code memory if a branch is taken (conditional and unconditional branches). The branch part of the execute stage can flush the complete pipeline in case a branch is taken. The flushes take care of the control hazards mentioned as dependency 6 and 7 on page 44.
The decode stage will decode the instruction and the relative or fixed offset for the branch address. The execute stage will determine if a branch needs to be taken (unconditional branches) and calculate the complete branch address. This branch address is communicated through a dedicated handshake channel to the fetch stage.
The fetch stage will change the PC (code memory address) and the next word read from the code memory will be fetched with the branch target address. The correct byte in the word gathered from the code memory will be marked with a *branch* flag. This byte will be sent throughout the pipeline including the *branch* flag and will therefore not be flushed in the execute stage. Normal operation resumes when the instruction with the *branch* flag enters the execute stage.

Interrupts can be addressed the same as branches, but are not implemented (yet).

MOV Operations
The move (MOV) operations are relatively easy. The decode stage is largely responsible for a correct behaviour of the MOV instructions, because this stage will read the operand which needs to be moved and address the register to which the data needs to be written. The MOV operations are therefore nothing more than a move from one memory address or register to another memory address or register. It is possible to move 16 bit data (e.g. move to the DPTR), but most data that is moved is 8 bit wide.
The same holds for extended move (MOVX) operations. The addressing of memory or registers is done the same as for regular moves.
The move instruction exists in the first four categories refer to [Table 17].

A special move instruction is the MOVC operation (move code byte). To eliminate some structural hazards the decision was made to communicate from only one place to the code memory. This need still exists and therefore we need to make an awkward mechanism for the MOVC instruction. The decode stage will detect the MOVC instruction and will store the current instruction address. The

execute stage will perform a branch to the MOVC address and the pipeline will be flushed (as is done for a normal branch) until the according byte is received. When the MOVC instruction is executed the execute stage will perform a second branch to the instruction address after the original MOVC instruction address, again everything will be flushed until the instruction after the MOVC instruction enters the execute stage. Normal program execution is resumed.

ALU Operations
The ALU performs arithmetic and logical operations. The ALU is a logic block that is spilt up in several separate blocks. Every separate block can perform an individual task, e.g. add, increment, swap of bits, etc. The correct task is indicated by the EB previously decoded by the decoder. It is possible to calculate 16 bit data (e.g. increment the DPTR), but most data that is calculated is 8 bit wide. Some ALU operations affect the condition flags in the PCON register (Carry, Overflow and Auxiliary carry flag).
The ALU is further split up in the following units:

- Shifting unit
- Logical unit
- Add unit (also for incrementing, decrementing and subtracting)
- Multiply unit

The fetch stage holds a separate adder for the program counter (PC). This avoids sharing of the ALU adder and eliminates the ALU hazard (refer to dependency 5 on page 44).

BIT Operations
A bit operation is only possible for a small part of the data memory (refer to [Figure 8]). It works, as a read-modify-write, on byte level which means that the read phase fetches a complete byte from the data memory and extracts the bit from this byte. After the bit manipulation it will write back the complete byte with the changed bit.
The bit manipulation can be several different things like comparison, logical OR function, logical AND function, logical Exclusive OR function, clearing and setting the bit.

Flush Mechanism
The flush mechanism is introduced to correctly deal with branch and MOVC instructions. After a branch the complete pipeline is filled with redundant instructions. These instructions needs to be cleared and will therefore be flushed in the execute stage (refer to dependencies 6 and 7 on page 44). The execute stage will not perform any writes to any register or memory at this point. This way the data remain valid in these registers and memory. The flush mechanism is stopped when the correct flag is read in the execute stage transferred from the previous stages.

5.5. Datapath

The datapath of a pipelined microcontroller is more complex than the datapath of a sequential design. Not only because of the need of extra registers between the pipeline stages, but also because of extra communication between frequently used registers R0..R7.
The need for extra registers is inevitable due the pipelined architecture. The extra registers hold temporarily variables between the stages of each instruction in the pipeline. These registers not only carry instruction data from one stage the next stage, but also control data. Any value needed in a later pipeline stage must be placed in such a register and copied from one pipeline stage to another until it is no longer needed. Some auxiliary registers used in other designs (HT80C51-LC and HT80C51-SU) will otherwise be overwritten before all uses of the registers are completed.
The HT80C51-HS will be fetching 32-bits from the code memory. This 32-bits information needs to be decoded in a correct way to maintain a correct behaviour for an 8-bit microcontroller. The HT80C51-HS does not have an internal data bus (the IDB like described in section 3.1), instead it uses dedicated channels which communicate between the independent stages and transfer the data from one register to another register in another stage.

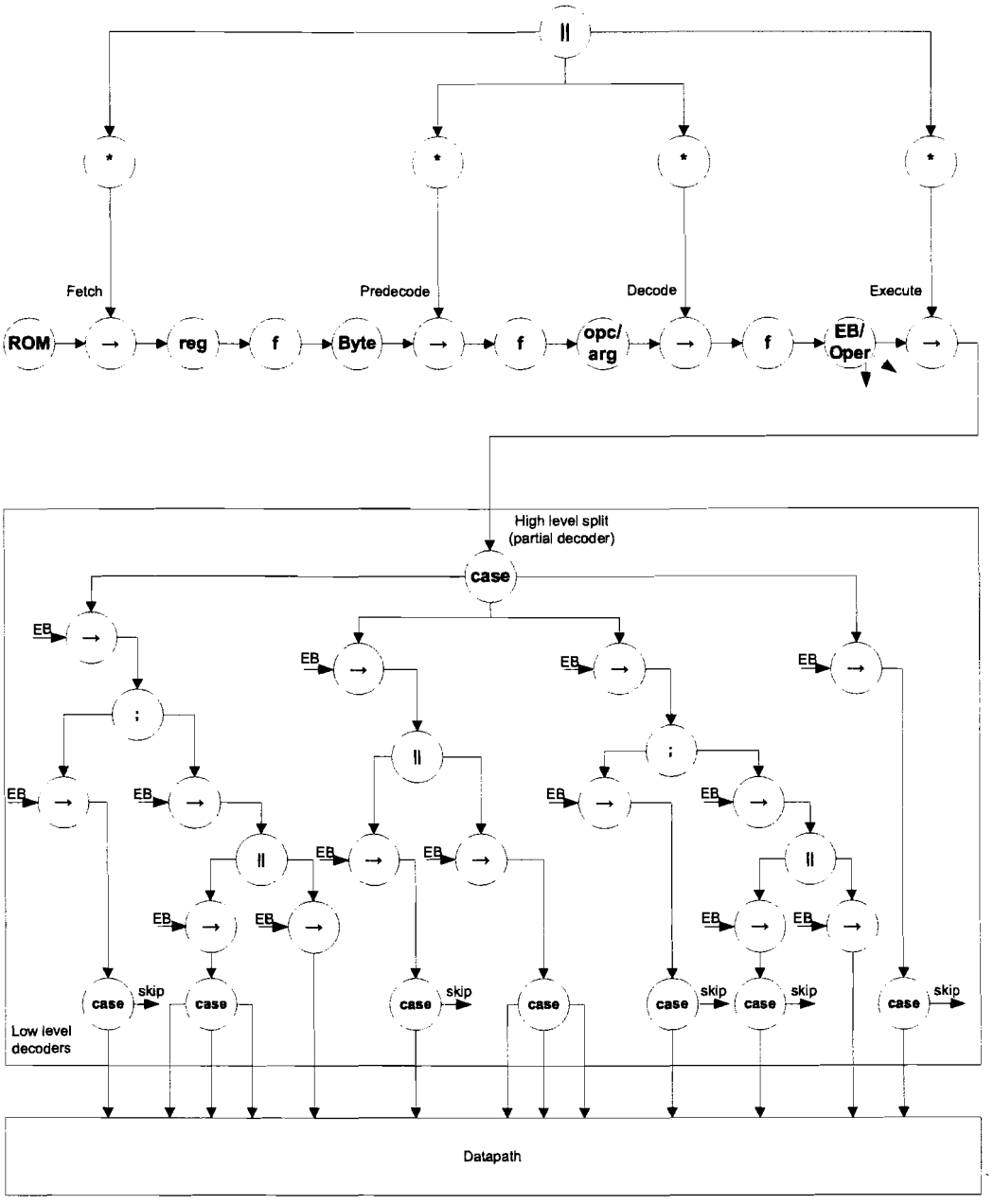
5.6. Control structure

The control logic of a pipelined design will be completely different compared to a non-pipelined design. Not only because the control logic will drive the independent pipeline stages and controls the logic function of these pipelined stages, but also because the control logic has to detect possible hazard situations that were not present in non-pipelined designs of the HT80C51.
The handshake channels between all four stages, memories, registers as well as the feedback channel to the fetch stage in case of a branch take care of a correct behaviour through the pipeline stages. Due to the control overhead each channel generates, it does not mean that every extra stage and therefore extra channel generates a boost in performance.

The control in the fetch stage is relatively straightforward. The main task of the fetch stage is to fetch instructions with its arguments at a rate that is fast enough for the decode stage and execute stage. While the fetch stage fetches 32-bits at once, it still decodes in the pre-decoder one byte at a time. It is imaginable that the 32-bits buffer in the fetch stage runs out of bytes and that the fetch stage first needs to access the slow code memory for new data. Until this time the fetch stage can not send a new instruction to the decode stage and everything can stall. Therefore, the fetch stage consists of two 32-bits registers which are controlled in a wagging way (refer 5.4.1). This process is completely independent of the second process in the fetch stage, the pre-decoding of an instruction and the channel communication to the decoder. The pre-decoder collects all data needed for the instruction and communicates all data to the decoder.

The decode stage will run in a sequential mode. It will decode the read and write bits needed to access the memories or registers. After that it will calculate if there is a conflict between the current instruction which is being executed and the next instruction in the pipeline. This scheduler controls the way the pipeline is executed. It is possible that the execute stage writes back calculated data to memories or registers while simultaneously the decode stage will read from other memories or registers. The decode stage will control the execute stage through the dedicated handshake channel called *fetch_to_decoder*.

The execute stage consists of one input channel from the decoder and two output channels for controlling the correct behaviour for the data. Data needs to be written back to the memories or registers by a channel. The conditional channel which only works when the execute stage calculates a branch address.
The control structure of the HT80C51-HS is organized as shown in [Figure 26]. The high level decoder splits up the control flow like the HT80C51-SU.



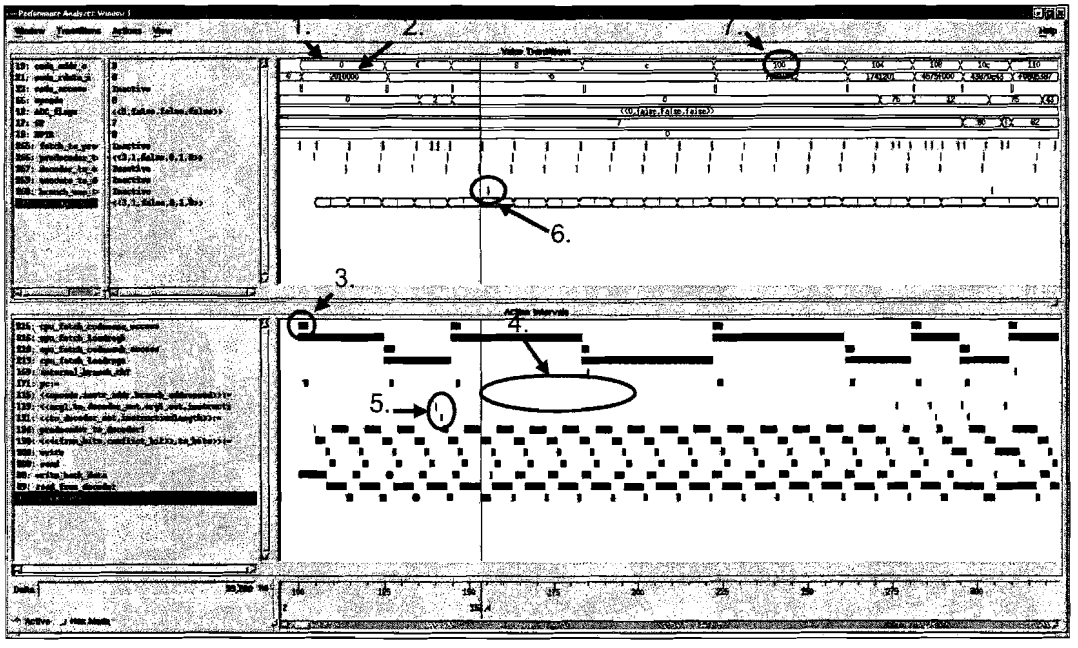
[Figure 26] HT80C51-HS handshake circuit

5.7. Optimizations

The complete 80C51 instruction set is implemented in the HT80C51-HS microcontroller. Some measurements are being done and the architecture will be analyzed to find bottlenecks. Some optimizations can be made to the design. These can be transformations for one goal only (obtaining higher performance, lower power consumption or a smaller area), but also so called macho transformations (term introduced by Andrew Bailey). Macho transformations are beneficial to the complete design in respect to power, speed and area.

5.7.1. Branch instructions

Some typical instructions are being executed which test the natural dependencies in a pipelined microcontroller design (previously discussed in 5.2). One of the structural and control hazards which can occur happen in a regular branch operation.
In [Figure 27] a LJMP is executed from address 0 (code_addr_o, see number 1). As can be seen 4 bytes are being fetched simultaneously (code_rdata_i, see number 2) 02010000h. The first byte; 02h is the opcode (LJMP which is a three byte instruction). The next 2 bytes are the arguments 0100h. The complete instruction LJMP 0100h will perform a long jump to code address 100h.



[Figure 27] HT80C51-HS branch operation

The total instruction sequence is as follows:

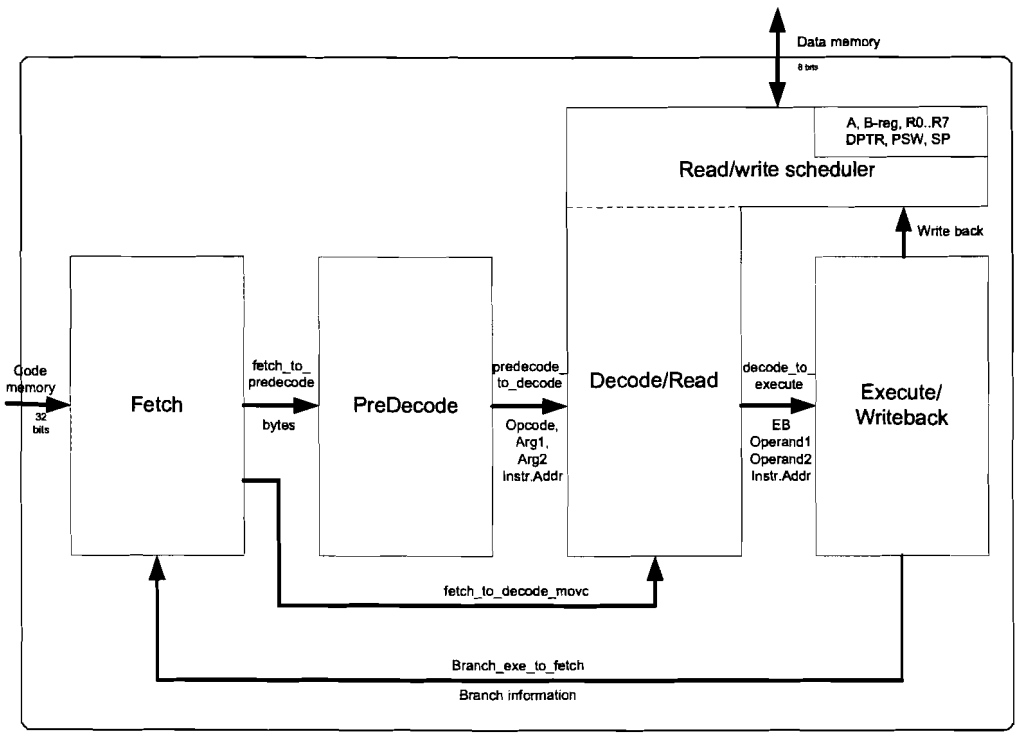
- Bytes are stored in the internal buffers in the fetch stage (see number 3);
- One byte each time is communicated to the predecode stage (see number 4);
- For a single byte instruction the byte gathered in the predecode stage is automatically the complete instruction. For the LJMP instruction the successive bytes are also gathered (see number 5) before the instruction is send to the decode stage;
- The decode stage will decode the EB, read and write bits. Then the read process will gather all operands needed (in case of the LJMP the operands are the arguments in the instruction) before they are communicated to the execute stage;
- The execute stage will perform the jump communication to the fetch stage (see number 6)
- This communication will be buffered in the fetch stage to clear the communication with the execute stage. The execute stage will flush every new instruction until it receives the branch target instruction;

- The fetch stage will change the PC (and therefore the code_addr_o, see number 7) fetch the word with the correct byte. The predecoder will extract the correct opcode from this word and collect the complete instruction which will be coloured for the execution stage (the execution stage will detect the correct colour corresponding with the correct branch). This will be communicated to the decoder;
- The decoder will decode all necessary bits and read the correct operands. This information is communicated to the execute stage;
- The execute stage will detect the colour in the instruction and stops flushing and start executing normal behaviour.

5.7.2. MOVC instructions

As can been seen in the total sequence a branch is long. However we need the complete flush mechanism to prevent hazards and we choose only one mechanism for branching (e.g. no branch prediction). Otherwise it is already possible for the predecoder or decoder to perform a branch in case of unconditional branches. This choice was made to keep the control overhead as small as possible. The MOVC instruction is also executed as a branch condition (as explained in 5.4), but with an even less optimized implementation. The MOVC will perform a double branch. To increase throughput, the branch and MOVC instruction should be optimized.

This is done by adjusting the flushing mechanism. The instructions need to be flushed in an earlier stage of the pipeline. It is not necessary that the instructions ripple through the complete pipeline, are decoded, and possibly access the memory for reading the operands.
When the conditional channel communication from the execute stage to the fetch stage finishes, the fetch stage marks that data can be flushed in the predecode stage. The predecode stage will flush all data from the fetch stage until it receives the branch target. This will be communicated to the decode stage. The advantage of this approach is that the predecode stage can flush faster than the execute stage. The fetch stage itself will flush as much as possible, without introducing any hazards. These two mechanisms also have a power consumption bonus.

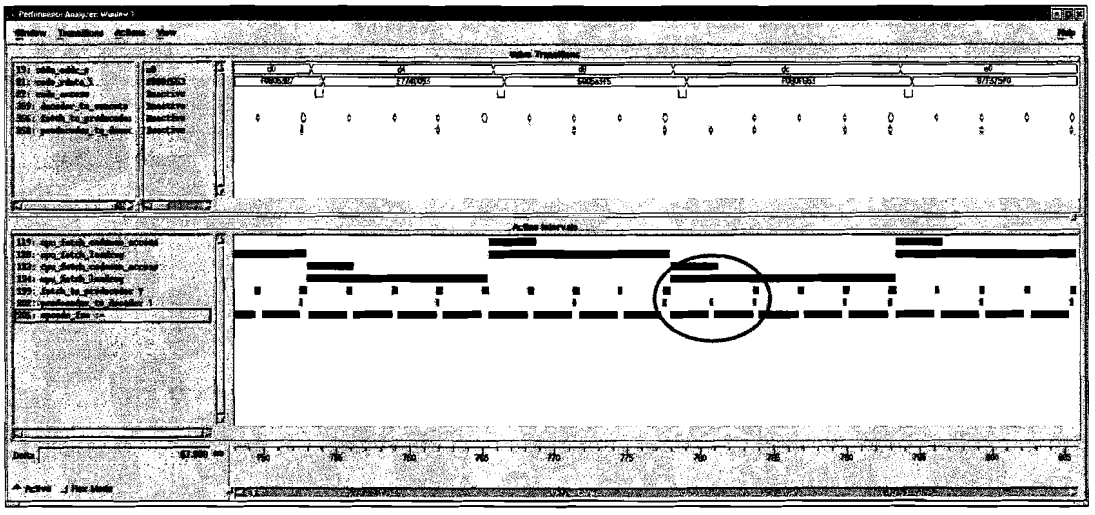


[Figure 28] HT80C51-HS global architecture with MOVC addition

The MOVC instruction also benefits from the improved flushing mechanism of the branch instructions, but it could benefit even more from some other architectural improvements. This section describes these architectural improvements.
The current implementation of the MOVC instruction executes like a double branch operation. First it will branch to the code address calculated in the MOVC instruction itself. Second it will branch to the original code address after the MOVC instruction. The second branch is when the instruction finishes and the byte from the code memory is successfully moved to the accumulator.

The architecture of the HT80C51-HS is set up such that all the operands of the instructions are known at the beginning of the fetch stage. The idea is to split up the MOVC instruction into 2 instructions. The first part executes a JMP instruction which is executed normally, but without the flush mechanism through the pipeline.
The second part executes a regular MOV instruction which gathers the code memory operand at the normal place in the pipeline; the decode/read stage. This can be seen as simplification of a complex instruction or implementing a multi-cycle instruction.

The disadvantage is that the architecture needs to be expanded by a channel communication and some added control logic in the fetch stage (see [Figure 28]). The implementation of the multi-cycle instruction is done in the predecode stage. This stage can now output more instructions than it receives from the fetch stage (see [Figure 29]).



[Figure 29] HT80C51-HS MOVC instruction as a multi-cycle instruction

First the predecoder outputs a MOVC instruction with an additional flag. This indicates that the execute stage should do a branch to the fetch stage. This branch will be marked as a special MOVC branch. The PC will not be changed and continue following the normal program execution flow. The branch buffer (see [Code Fragment 26]) is expanded with control logic to access the code memory and to communicate the result gathered from the code memory to the decode/read stage by the new dedicated channel. The new access to the code memory is arbitrated by a special arbitration structure in Haste (see [4]). Therefore it will not create a new hazard or dependency.

Second the predecoder communicates a MOVC instruction without the additional flag to the decode stage. This indicates that the decode stage should wait until it receives the byte communicated by the new dedicated channel from the fetch stage. When this communication ends, the result is sent to the execute stage and stored (by a MOV instruction) in the accumulator.

After the MOVC instruction the program executes regularly. No flushes are being executed.

```
fetch =
[
  (   load_reg_1() || empty_reg_2()
    ; load_reg_2() || empty_reg_1()
  )
  ||
  buf_branch_channel()
]

buf_branch_channel =
[
    branch_exe_to_fetch?buf
  ; if buf.movc then
        set_codemem_addr_access()
      ; fetch_to_decode_movc!code_rdata
    else
      internal_branch!buf
    fi
]
```

[Code Fragment 26] HT80C51-HS Pseudo code of the fetch stage with MOVC addition

5.7.3. Indirect addressing for the registers

Some of the special registers (the accumulator or R0..R7) are accessible in a direct way (e.g. INC A or MOV A, R0). These frequently used registers are residing locally in the decode/read stage instead of the data memory. For communications to these frequently used registers we do not need to access the data memory bus by complicated procedures, but we can address them locally through their own channels.
The special registers are also accessible through an indirect addressing mode, e.g. INC E0h. Address E0h is the address of the accumulator and performs the same operation as INC A. The disadvantage is that the indirect addressing modes will start up the shared memory access routine. This complicated routine will eventually notice that the addressing is not meant for the data memory but for the internal register. The access will be rerouted to the internal registers. This structure will share a complicated memory access routine and share the access to the heavily used internal registers.

The solution for this problem is to decode the addressing already in the decode stage and let it communicate with the correct procedure. With this structure there is less sharing on the communication channels to the internal registers and the memory access routine.

5.7.4. Registers in the execute stage

The frequently used registers (e.g. A, B, R0..R7, DPTR, etc) are now residing in the decode/read stage. This has many advantages when reading the registers but in case of writing back, a channel communication will be needed to write back the data to the registers. Also to avoid hazards the registers should be scheduled by a specific scheduler in case of reading and writing the registers.

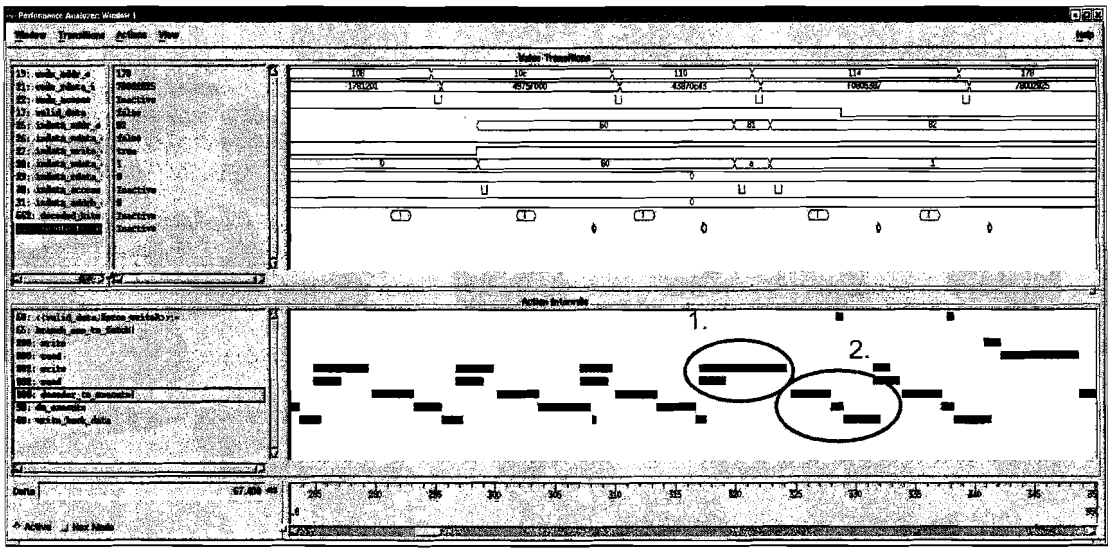
The control and scheduling is relatively slow and complicated. It could be done easier if the execute stage is in control of the registers. The decode/read stage will only decode which register needs to be read and which register needs to be written. The execute stage can control the data in the registers and the validity of the registers.

The combined decode/read stage will be a normal decode stage with a possible memory read process (code memory, data memory, extended data memory, SFR memory) to gather the memory operands for the execute stage.

5.7.5. The fifth stage: decoupling the write back

Section 5.7.4 explains that the frequently used registers are shifted from the decode stage to the execute stage for a simplified arbitration scheme (no arbitration needed for the registers because they are read and written in the same stage) and to avoid communication. The only write back which is special and needs to be taken into account is the relatively slow memory write back. This remains a special task (like memory read in the decode stage) with a dedicated memory access routine.
When memory conflicts (structural hazards or data hazards) are absent it is logical to make the memory write back a separate process which takes care of the memory access. A memory conflict is dependent on the program code executed on the microcontroller. The instruction following the instruction which writes to the memory is responsible for the memory conflict.

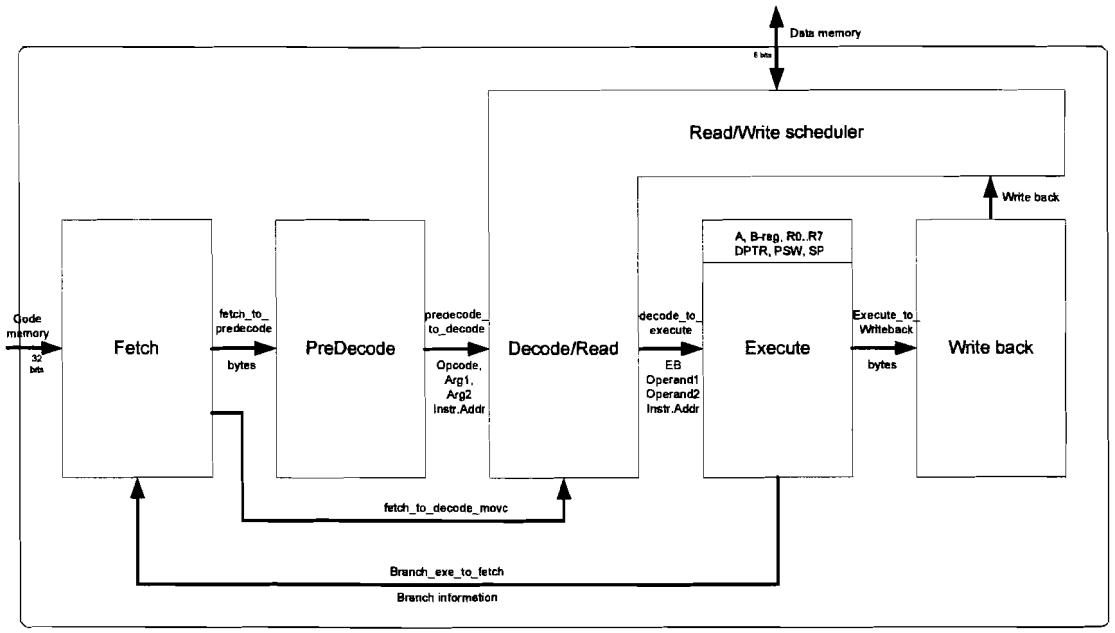
The advantage of decoupling the write back from the execute stage is that the write is a fully autonomous process which can be independent of the execute stage. Now the decode stage and the execute stage will wait until the write back (see [Figure 30] number 1) is finished before they continue operation (see [Figure 30] number 2). This is because the write back process is in sequence with the actual execution of the instruction. And the decode stage waits in all cases on the feedback of the channel communication from the execute stage. When decoupling this fixed structure the decode stage (with the scheduler) will only wait on the write back if there is write back being planned to be executed or actually being executed.
The decode stage and execute stage will become more free running when decoupling the write back.



[Figure 30] HT80C51-HS write to the data memory

The scheduler in the decode stage prevents the control of structural and data hazards (refer to dependency 2, 3, 4 and 15 on page 44-45).
The memory write back will be a separate stage running independent and in parallel with the other four stages. The write back to the memories will be started by the execute stage. The correct scheduling of the instructions and memories is done by the read/write scheduler. The scheduler can insert stalls to the execute stage until the write back stage is finished with its communication to the memories.
There is no data forwarding implemented yet, but it is possible.

[Figure 31] shows the overview of the optimized HT80C51-HS architecture.



[Figure 31] HT80C51-HS global 5-stage architecture

5.8. Results

This section focuses on the measurements and results for the HT80C51-HS microcontroller after optimisation.

5.8.1. Implementation of the HT80C51-HS

The complete 80C51 instruction set is implemented in the HT80C51-HS and is fully compatible to the original 80C51 instruction set. The HT80C51-HS is validated with several testbenches which execute together more than 5000 instructions. All the numbers (power, speed, area) can be determined, but the numbers of power and speed are testbench dependable.
If a programmer or the assembler is familiar with the design of the microcontroller, a great increase in performance can be realised. In spite of this dependency, an analysis of the performance and power consumption will be made in the next paragraphs.

The HT80C51-HS is fully designed in Haste and executes at an average of 28.3 MIPS (for 5 ns memory access times). This is more than three times faster than the original HT80C51-LC design!

All previously mentioned suggestions as 32 bits fetching of instructions, pipelining and internal registers for R0..R7 are implemented successfully in the HT80C51-HS design.
After the optimization process the pipeline is split into five separate stages: fetch, predecode, decode, execute and write. These five stages communicate with dedicated handshake channels for save operation and correct synchronization between the stages.
The internal registers R0..R7, accumulator, DPTR, etc. can be accessed directly and indirectly through address recognition in the decode stage. Also different types of memories (SFR, internal data memory, external data memory) can be addressed in the correct manner.

Dependencies are recognised, analysed and dealt with correctly. Many dependencies are excluded due to the architecture of the HT80C51-HS, but also dedicated techniques are used to exclude

dependencies. The main solution to many of the data dependencies and structural dependencies is the arbitration obtained by the read/write scheduler in the decode stage.

The interrupt functionality still needs to be designed and implemented in the HT80C51-HS. Implementing it in the predecoder stage like a new CALL instruction should not give any problems. The predecoder stage is already prepared for multi-cycle instructions (like the implementation of the MOVC instruction) and can be extended for the interrupt functionality.

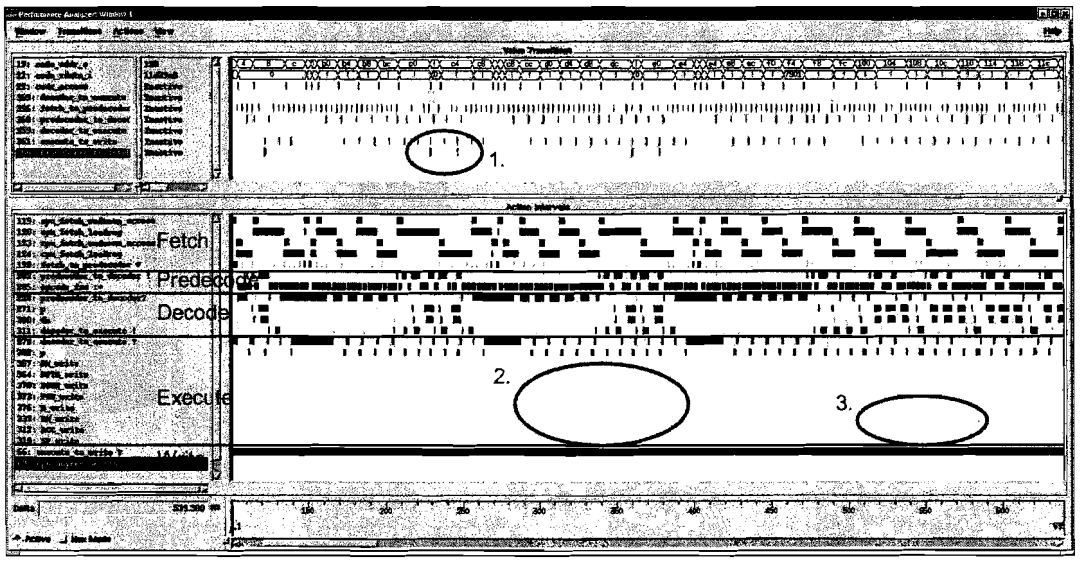
[Table 18] shows the results of the HT80C51-HS compared to the HT80C51-LC and the HT80C51-SU microcontroller. The 28.3 MIPS result is calculated from the same testbenches as used the HT80C51-LC and HT80C51-SU. All these measurement are done with 5.0 ns memories and a CMOS 0.14 µm technology.

	Performance		Area		Power	
HT80C51-LC	8.9 MIPS	100%	7705 gates	100%	68,9 pJ/Instr	100%
HT80C51-SU	12.5 MIPS	+40%	8230 gates	+7%	84,7 pJ/Intr	+23%
HT80C51-HS	28.3 MIPS	+218%	16840 gates	+119%	119,2 pJ/Instr	+73%

[Table 18] Performance, area and power results

5.8.2. Behaviour of the HT80C51-HS pipeline

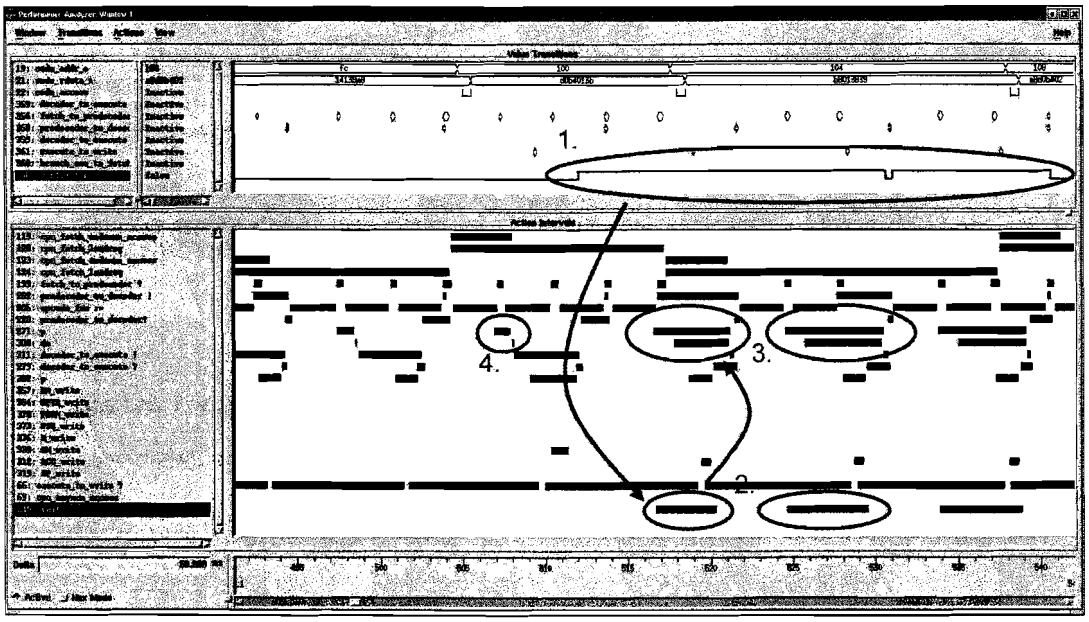
The HT80C51-HS consists of five pipeline stages. The blue lines in [Figure 32] separate the five stages.
The five stages run in parallel as can be seen in [Figure 32]. The fetch and predecode stage executes the processes previously described in 5.4.1 and 5.4.2. The decode stage receives the data from the predecode unit and calculates the EB, write and read bits. The execute stage receives the operands from the decode stage and will execute the instruction. In case of a branch instruction the execute stage will communicate the new PC to the fetch stage (see circle 1 in [Figure 32]). In case of ALU or MOV instructions the execute stage will execute the instruction and write back the result to the internal registers or to the write stage (see circle 2 and circle 3 in [Figure 32]).



[Figure 32] HT80C51-HS instruction execution

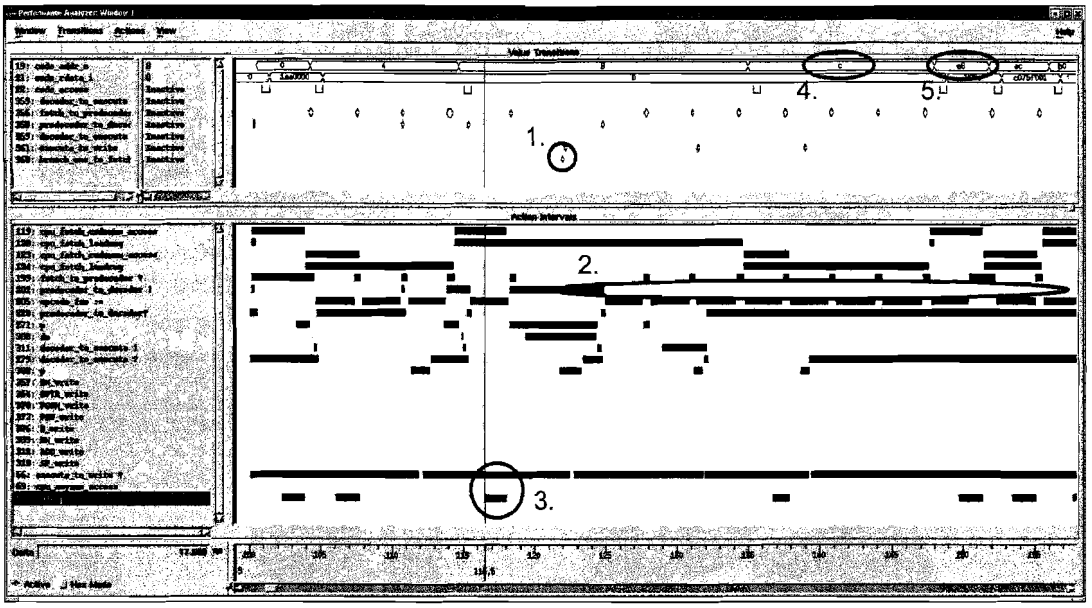
In case there is not a conflict between the instruction currently being executed or written back and the instruction in the decode stage, the read process is being executed.

When there is a conflict between the instructions; the write back stage wants to write to the data memory which the next instruction wants to read from in the decode stage, the read access is postponed until the write access is finished by the read/write scheduler discussed in 5.4.4. The write back stage needs to communicate to the decode stage when the actual write to the data memory is finished.
This can be seen in [Figure 33] where the read/write scheduler indicates there is a conflict (see circle 1) and the actual reading of the memory is postponed (see circle 3) until the write stage communicates to the decode stage (see circle 2). This prevents hazards to the data memory (refer to dependency 2, 3, 4 and 15 on page 44). As can be seen in circle 4 the read access is much faster without the conflict.



[Figure 33] HT80C51-HS instruction execution with dependent read/write

The branch executions are a bit different from the regular instructions. The decoder detects if an instruction is a branch or regular instruction. The execute stage determines if a branch should be taken or not (conditional branches). If a branch needs to be taken, the new PC is being calculated by the execute stage and this information is sent to the fetch stage by a dedicated channel. This is shown in circle 1 in [Figure 34]. This will flush the pipeline and bytes in the internal registers in the fetch stage and prevents further communication to the decoder stage (see circle 2). Because the PC is already updated before the fetch stage is aware of the branch (see circle 3), the fetch stage will once more fetch a redundant byte (see circle 4). As seen in circle 5 the PC (code_address) is changed into the correct address for the code memory. Correct program behaviour continues.



[Figure 34] HT80C51-HS instruction execution with branch

5.8.3. Analysis of the measurements

Section 5.8.1 includes the measurements for HT80C51-HS. The core is completely implemented and all instructions are executed for the measurements.
To analyse the pipeline stages of the HT80C51-HS, each stage is simulated separately to measure the speed and analyse the behaviour of each stage.
The speed of each stage is described below. Dot the stages a minimum and a maximum value is presented. Due to different instructions different speeds can be obtained, the stages are so-called multi-rate.

The performance of the HT80C51-HS is analyzed by the interactive performance analysis tool: Handshake Technology Profiler (htprof). This tool finds and reports the slowest timing path between two signal transitions at a given time. It is based on simulation and the found paths can be different from instruction to instruction. Therefore it is important to use the correct testbench and analyse the correct timing interval.

[Table 19] shows a summarised table of the individual average speed of the stages.

The fetch stage is a very small and fast stage. The setup with wagging buffers reaches a speed of 125 MHz stand-alone. This is without interrupts by the MOVC or branch instructions.

The predecode stage can operate between the 24 MIPS for a three byte instruction and 98 MIPS for a single byte instruction. This all depends on how many bytes the predecode stage should collect from the fetch stage to gather a complete instruction with its arguments. An average of 63 MIPS is achieved by benchmarking it with the regular testbenches.

The decode stage operates at 15 MIPS, 22 or 41 MIPS. This time is variable because it is instruction depending on the number of data memories reads. It is possible that the decode stage should read twice, once or even not from the data memory.

The execute stage will also execute instructions at a variable rate. Not every instruction needs the same time to be calculated. The MOV instructions are relatively fast compared to an ADD or a complicated MUL instruction. The MOV only needs to move a byte from the correct input to the correct output which are already decoded and assigned by the decoder. The ADD or MUL (but also other

ALU) instructions need a real execution before the results are moved (or transferred) to the correct output. The branch instructions not only need the ALU, but also a communication to the fetch stage by a dedicated handshake channel. The fastest instructions operate around 45 MIPS (like MOV) while the slowest (like MUL) executes at 11 MIPS. The average is 28.3 MIPS.

The write back stage is fairly easy. It is a stage which is not always being executed. When it executes it will do only two things: receive from the execute stage and write to the data memory. It is possible that the write back stage sequentially writes two times to the data memory. This is necessary for the CALL instruction which needs to write two bytes to the memory (indicated by the SP). In this case the throughput of the write back stage will be 19 MIPS. For a single write back it will be 34 MIPS and when write backs are absent this stage is not being executed.

It is difficult to pinpoint a single bottleneck in an asynchronous pipelined design. For some instructions the pipeline can be truly balanced, while for other instructions a single stage in the pipeline can be a bottleneck. In general the most occupied stages are the decode stage (with the memory read accesses) and the execute stage.

	Fetch stage	**Predecode stage**	**Decode stage**	**Execute stage**	**Write back stage**
	MIPS	MIPS	MIPS	MIPS	MIPS
HT80C51-HS	125	63	31	28.3	34

[Table 19] HT80C51-HS individual average performance of the stages

There are three major changes in the HT80C51-HS architecture:

1. Pipelining
2. 4 Bytes fetch for code memory
3. Reside frequently used RAM space in locally accessible registers

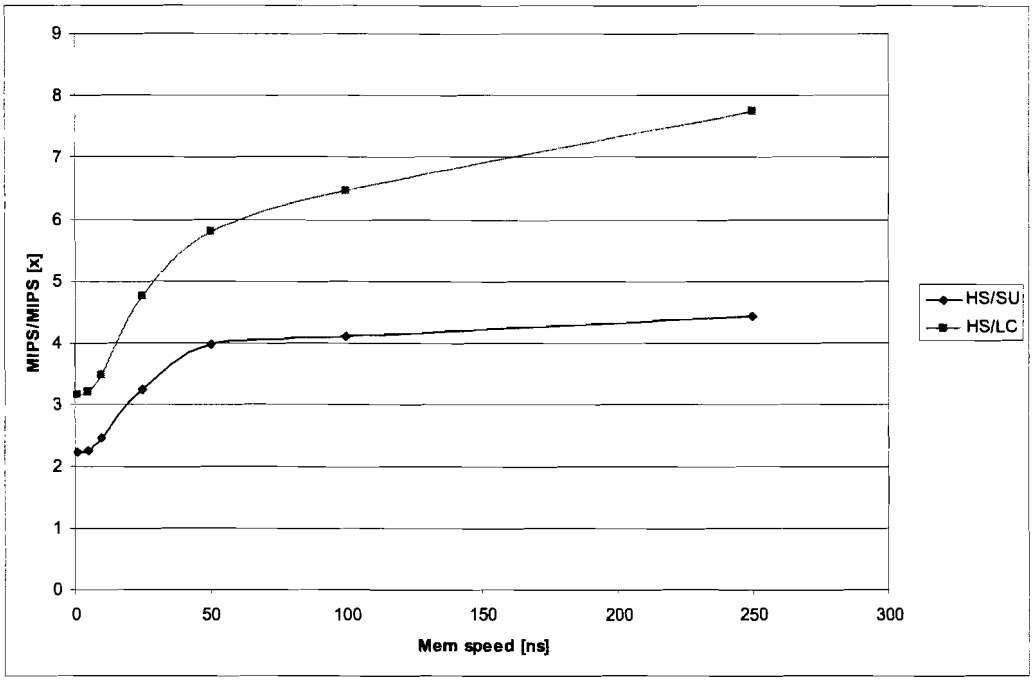
The influence of fetching four bytes at once for the code memory can have bigger influence with a slower code memory. Until now all the measurements are done with 5 ns code memory and data memory. Three experiments are done:

1. Variable memory speeds for both the code and data memory for the HT80C51-LC, HT80C51-SU and HT80C51-HS (see [Figure 35]).
2. Variable data memory speed with a fixed code memory speed for the HT80C51-SU and the HT80C51-HS (see [Figure 36]).
3. Variable code memory speed with a fixed data memory speed for the HT80C51-SU and the HT80C51-HS (see [Figure 36]).

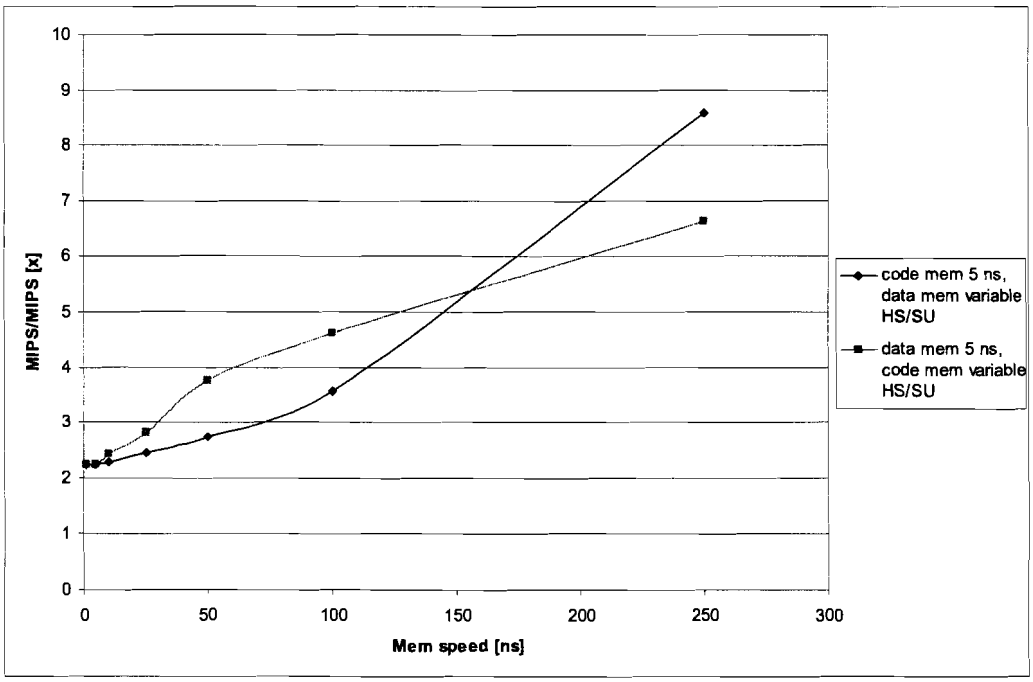
As expected the slower memory speeds have less influence on the HT80C51-HS than on the HT80C51-LC or HT80C51-HS. Comparison between the HT80C51-HS and HT80C51-LC (see [Figure 35]) shows that the performance factor can easily increase to almost 8 times. This gain in factor is not only due to the four bytes fetching, but also due to fewer data memory accesses. This decrease in data memory accesses is due to the third design decision (reside frequently used RAM space locally inside registers).

To measure the influence of the design choice to fetch four bytes parallel from the code memory another experiment is done. [Figure 36] shows the result of this experiment. The first part of the experiment is that the data memory has a variable speed and the code memory is fixed speed at 5 ns. The second part of the experiment is that the data memory has a fixed speed at 5 ns and the code memory has a variable speed.
The red line shows almost the same graph as [Figure 35], while the blue line shows a different path. The advantages of the multiple bytes fetch in the code memory of the HT80C51-HS and the fewer data memory accesses will become clearer when only changing the data memory speeds. The blue line inclines steeper than the variable memory speeds (see [Figure 35]). The data memory from 1ns to 250 ns makes almost no difference in speed performance of the HT80C51-HS and causes the steeper incline of the blue line. The performance of the HT80C51-SU will decrease much more due to more

data memory accesses than the HT80C51-HS. In the case of a fast data memory (red line) and slow code memory the advantages are still there, but smaller.



[Figure 35] Comparing HT80C51s performing with variable memory speeds



[Figure 36] Comparing HT80C51s performing with fixed and variable memory speeds

5.9. Limitations of the Handshake Technology flow

As remarked in this thesis before, some parts in the Handshake Solutions design environment (TiDE) can be improved. This chapter presents a few of these improvement areas.

5.9.1. Access to the memory

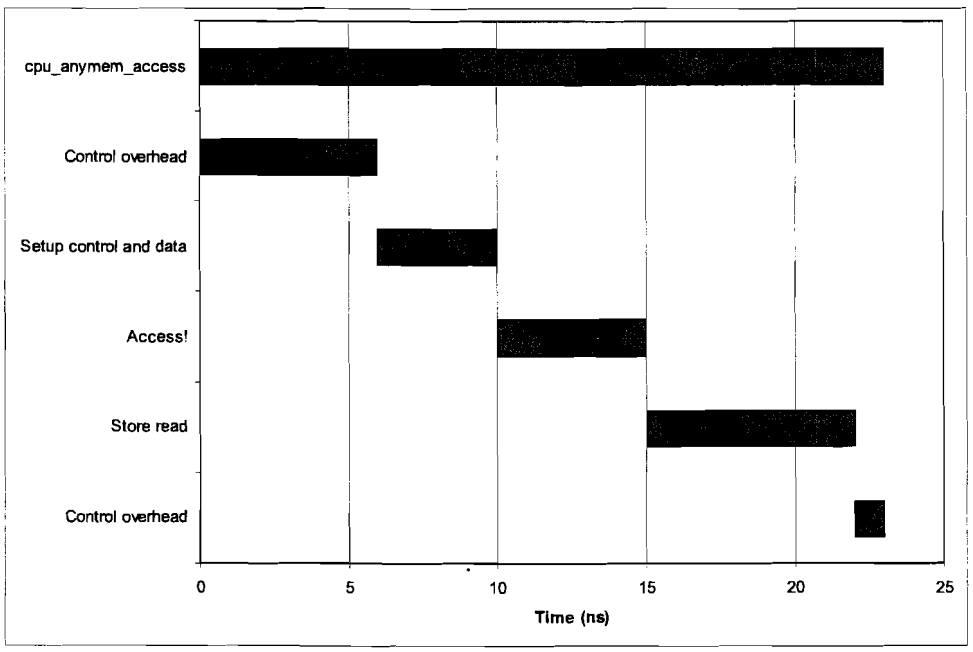
The standard routine to access the memory as mentioned in chapter 3.5 uses the following steps: assign the calculated address to the memory address, do the actual memory access and finally read (or write) the data (see [Figure 38]).
The total process takes 19 ns to complete for the write (see [Figure 38] circle 1.) and 23 ns to complete for the read (see [Figure 38] circle 2.), while the actual memory access takes only 5 ns to complete.

The complete routine to access the data memory becomes so slow due to the sharing overhead (the cpu_anymem_access can be accessed twice on two different places: two times in the decode/read stage and two times in the write stage).

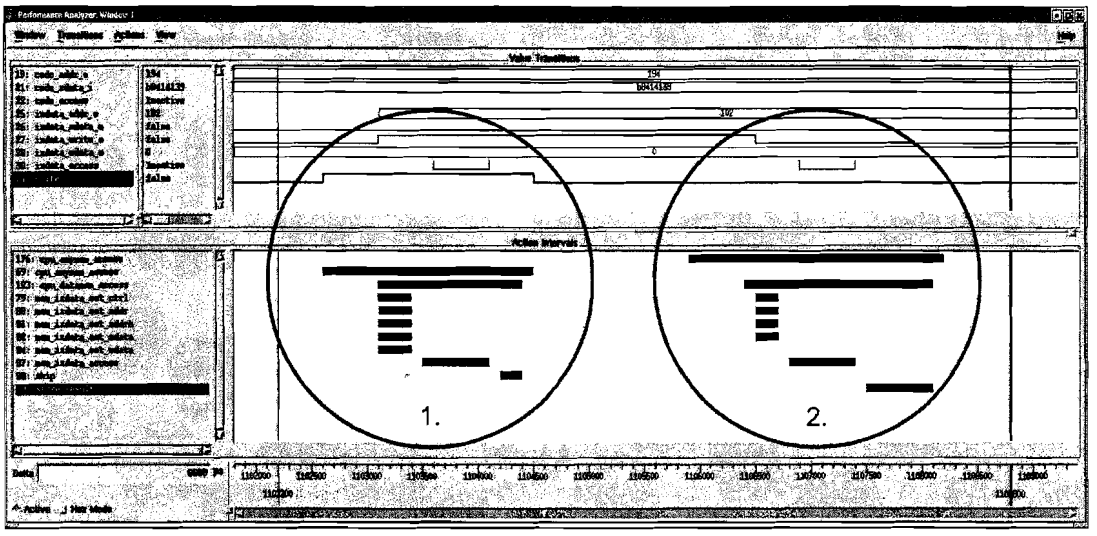
[Figure 37] shows the timing diagram of the data memory access. This total access access (cpu_anymem_access) exists of:

1. Control overhead
2. Setup time of control and data
3. Actual access memory access (of 5 ns)
4. Storage of the data (for a read access)
5. Control overhead

When there are many accesses to the data memory, the penalty for a memory access is too high. This problem is smaller with slower memories.



[Figure 37] HT80C51-HS timing diagram for data memory access



[Figure 38] HT80C51-HS read and write data memory access

5.9.2. Handshake communication channels

One way to communicate safely between pipeline stages is to use handshake channels (refer to section 2.2.2). Handshake channels in Handshake Technology feature a four phase protocol and a single rail implementation. The advantages of using this type of implementation are that a single rail implementation is smaller, faster and uses less energy than the double rail counterpart.
Two phase handshake protocols have the same advantages; being faster and more power efficient (due to fewer gate transitions). The advantage of the four phase protocol implementation is robustness and mostly being smaller (two phase protocols can be more complex to implement and therefore larger).
It is not only necessary to implement faster handshake channels for communications throughout the pipelined stages, but also in processes inside the stages itself.
At this time experiments are ongoing to really test and measure the benefits of a two phase protocol instead of a four phase protocol.

5.9.3. Slow loops

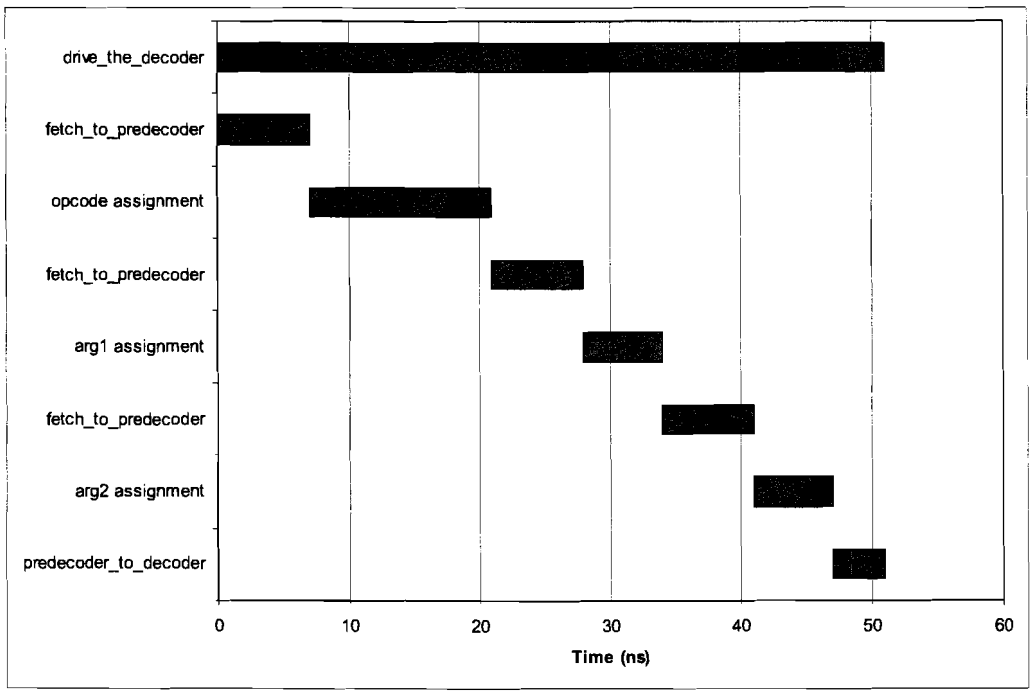
A good example that some parts in asynchronous technology are slow is shown in the predecoder stage. Analysis shows that the predecoder stage shown in [Figure 21] and [Code Fragment 22] can be the bottleneck for several instructions.
This process is an autonomous process, which communicates through two dedicated channels with its predecessor and successor, respectively.
The predecoder stage is a free running multi-rate stage procedure and consists of three sequential tasks as can be seen in [Code Fragment 27],

- First the input handshake channel communication, load_predecoder?<<byte, instructionaddress, branch>>, collects the input variables;
- Second task is an if-construct which places the byte which is sent by the input_registers into the correct register (opcode, arg1, arg2 or instructionaddress). If the first selection is taken the guards are calculated;
- Third task is the conditional handshake communication to the decode stage by the predecoder_to_decoder channel. This only applies when all registers are filled with the correct data.

This total sequence takes 51 ns when fetching 3 bytes which are immediate available from the two internal 32 bits registers. Three input handshake communications are necessary to collect the three bytes. One output handshake communication is necessary to send the fetched data to the decoder. The if-construct is executed three times: once for setting the opcode and performing the guard calculation, one for setting arg1, and one last time for setting arg2.

```
drive_the_decoder =
[
    fetch_to_predecoder?<<byte,instructionaddress,branch>>
  ; if branch_guard() + opcode_guard() then
        Opcode:=byte
      || calculate_guards()
    or arg1_guard() then
      Arg1:=byte
    or arg2_guard() then
      Arg2:=byte
    fi
  ; if sent_to_decoder_guard() then
        predecoder_to_decoder!<<Opcode,Arg1,Arg2,instructionaddress>>
      || calculate_guards()
    fi
]
```

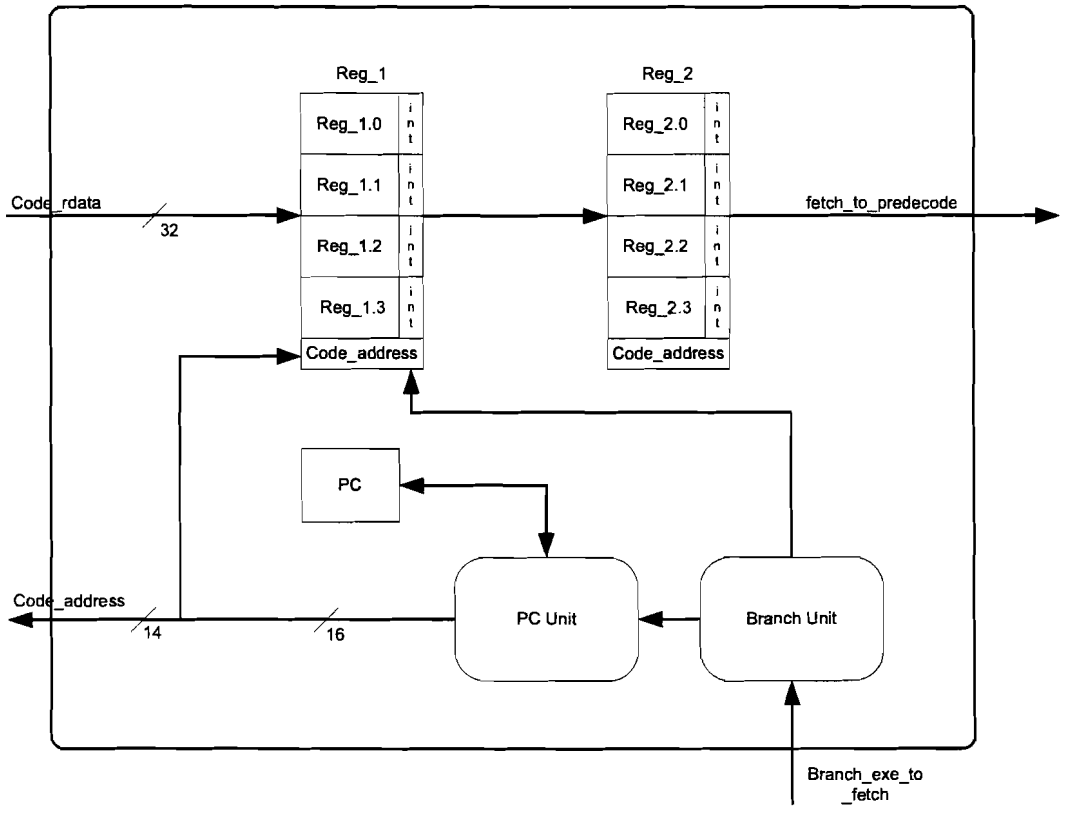
[Code Fragment 27] The execution of drive_the_decoder process



[Figure 39] HT80C51-HS timing diagram drive_the_decoder process in the predecoder stage

[Figure 39] shows the timing diagram of the drive_the_decoder process in the predecoder stage. This clearly shows that the channel communications from the registers (load_predecoder) takes 7 ns. That is twice as much as the channel communication to the decode stage (predecoder_to_decoder), which takes 3 ns. This is because the load_predecoder channel first leads through an eight selection multiplexer for the reg_1.0..3 and reg_2.0..3 bytes. Also the guard calculation takes much time in the opcode assignment.

A small experiment is done to view the extra added logic introduced by this eight selection multiplexer. The wagging buffers (refer 5.4.1) are replaced by two 4-byte buffers in series as can be seen in [Figure 40]. This adds an extra channel between these buffers (Reg1_to_reg2), but it reduces the logic on the fetch_to_predecoder channel.
Although there is a need for an extra channel communication this is a faster approach than the two 4-bytes buffers in a wagging setup. The reduced multiplexing hardware saves 3 ns for each load_predecoder call. When there is no extra communication needed between the two buffers this will be a 17% (9 ns) total profit.



[Figure 40] HT80C51-HS local experiment with load_input_registers process

6. Conclusion

This chapter is divided in the following sections:
Section 6.1 presents a summary of the results of the analysis of the synchronous 80C51, HT80C51-LP and the HT80C51-LC.
Section 6.2 outlines the results of the transformations presented in chapter 4 for the improved HS80C51-SU.
Section 6.3 describes the results from the new architecture HT80C51-HS in chapter 5, compared to the HT80C51-SU and the HT80C51-LC.
Section 6.4 presents recommendations for future work.

6.1. Results from the analysis

To gain proper insight in the 80C51 functionality the specifications of the synchronous standard 80C51s are analyzed as well as some synchronous high-speed versions of this architecture. The standard 80C51 has a machine cycle which takes twelve clock cycles. Instructions can take one, two or four machine cycles to execute. Some synchronous high-speed implementations consist of a better control structure where clock cycles in a complete machine cycles are reduced from twelve to six or even two and one clock cycle (e.g. NXP P89xx, CAST R8051XC, etc.). Other high-speed versions apply pipelining to the control structure and datapath (e.g. INTEL MCS51, Dolphin Flip8051-Cyclone, etc). The analysed synchronous, pipelined architectures consist of two, three or five stages and included multi-looped pipelines (complete or parts of the pipeline stages which need to be taken more than once to execute a single instruction).

The HT80C51-LP by Hans van Gageldonk is the first microcontroller designed in Haste. This architecture is designed to save power wherever possible. The second objective of this design was to keep the circuit area as small as possible. This was done to reuse as much of the datapath as possible. This applies to registers, communication paths and arithmetic logic.

The H80C51-LC is the second generation of the HT80C51. This design is the starting point for this thesis. The HT80C51-LC is a fully sequential 80C51 architecture which leads through the five phases of executing an instruction: fetch, decode, read, execute and write. The analysis of this design identified several speed bottlenecks. For example, the usage of memory was high (some instruction accessed the code and data memory more than five times) and every memory access shows a high performance overhead. The latter was partially caused by multiplexing the different memories on multiple locations in the architecture. Also the architecture was sequential and the actual execution of some complex instructions had significant overhead in the control structure.

6.2. The HT80C51-SU

The HT80C51-SU presents an initial speed up of the HT80C51 microcontroller. The goal of this design is to improve the HT80C51 performance without a major architectural change. The control structure was changed to tackle the control overhead problem in the HT80C51-LC design. Instructions are divided in four main parts by a high-level decoder (MUL/DIV instructions, branch instruction, MOV instructions and the other instructions).

This approach also allows the introduction of a pre-fetch unit. Some instructions which do not need the result of the current instruction can already start the pre-fetch of the next instruction.
Another change is that some individual instructions are optimised, e.g. the execution of the divide and multiply instruction became much faster due to a different implementation of the calculation.

The combined changes resulted in a speed up of 40%, area increase of 7% and an increase in power of 23% (refer to [Table 18]).

6.3. The HT80C51-HS

The thesis work presented in this report is a positive step forward in the search for more speed in asynchronous designs made by the Handshake Solutions design flow. The HT80C51-HS is designed fully in the Handshake Solutions' design language, Haste. The complete functionality of the original 80C51 is implemented in the HT80C51-HS. All 255 instructions are implemented and validated.

The HT80C51-HS is now performing at an average speed of 28.3 MIPS (worst-case conditions). This improvement of a factor 3.2 is mainly due to the three major changes:

1. Pipelining
2. 4 Bytes fetch for code memory
3. Reside frequently used registers locally

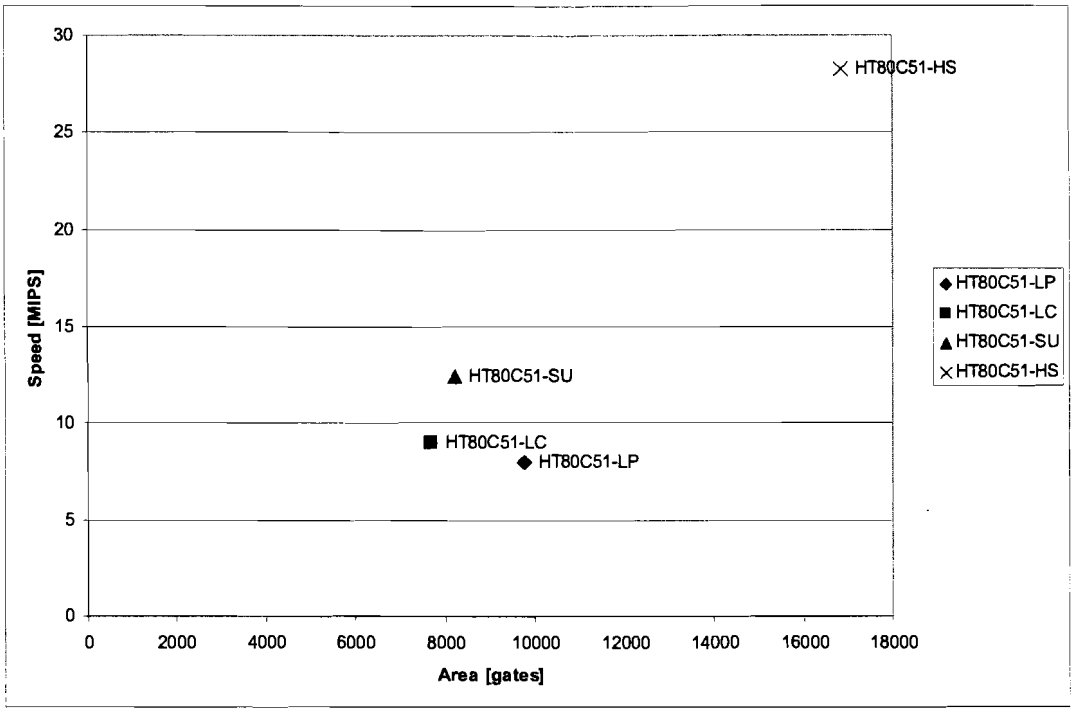
These three changes are implemented in a single iteration, therefore it is difficult to say which change has the most impact. Despite of the five stage pipeline the performance improvement is not a factor 5 (at least not with these fast memories). This is due to complicated control, control overhead due to more channel communications between the stages, longer combinational paths and arbitration for the different type of dependencies (see page 44).

[Figure 41] is showing a graphical overview of the different asynchronous 80C51 implementations made by Handshake Solutions. [Figure 41] shows that the HT80C51-HS outperforms in speed every other HT80C51 made earlier. The HT80C51-HS shows a performance boost of a factor 3.2. This performance increase comes with a penalty for area. This penalty is a factor 2.2 times, but the HT80C51-LC is designed with low cost as main design goal.

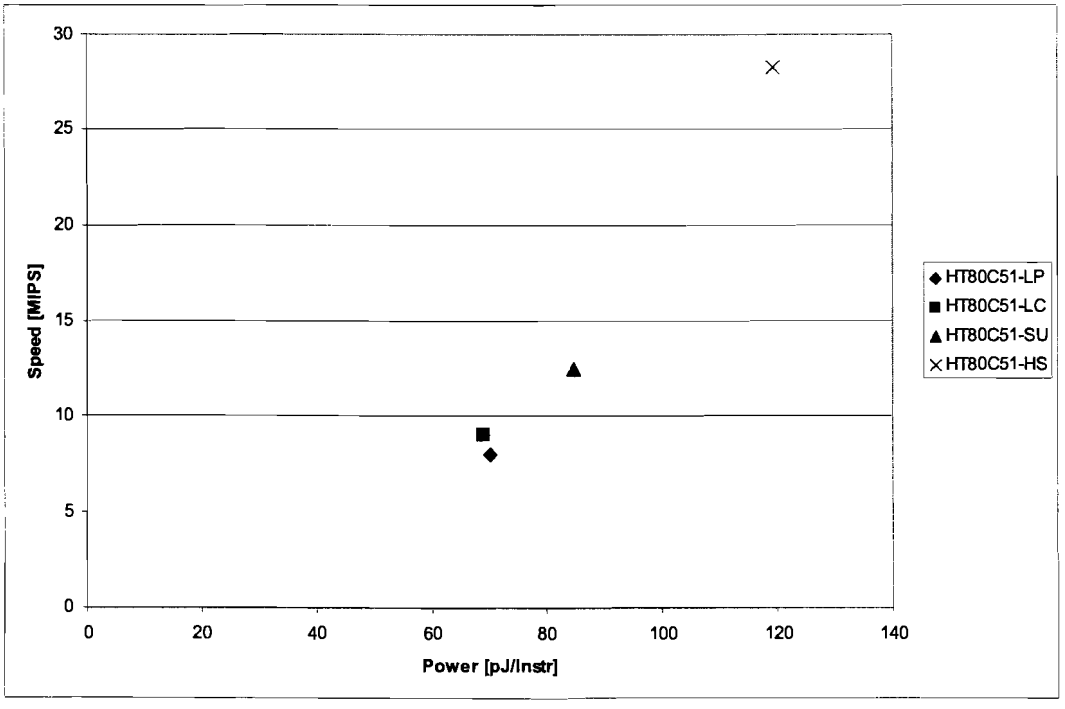
[Figure 42] shows the second disadvantage of the HT80C51-HS architecture. This is the increase in power consumption. The power penalty is a factor 1.7.

The memory time experiment (variable memory speeds) shows that fetching four bytes in parallel works at fast memory speeds (5 ns). The main advantage of this design choice is with slower memories. The parallel fetching of the bytes offers an internal buffer without accessing the code memory multiple times.

The pipeline is not perfectly balanced at this point. It is possible to merge the fetch stage and the predecode stage to reduce the control overhead. It is not sure that this architecture change will increase the instruction throughput, but it will reduce the area overhead.
To increase the instruction throughput the execute stage and the decode stage needs an optimization. The most critical stage is the execute stage, but this is instruction (code) dependable.



[Figure 41] Graphical overview of the HT80C51 microcontrollers in terms of speed and area



[Figure 42] Graphical overview of the HT80C51 microcontrollers in terms of speed and power

6.4. Recommendations

In this section a few suggestions are given to extend the system and possible increase system performance. These ideas arose while designing the HT80C51-HS, but were not implemented due to limited time available for this thesis. These ideas may be interesting to investigate in more detail or could be implemented in the future.
The improvements, which can be made to the architecture, do not always result in an increase of the performance. These are all suggestions which, when implemented, should be carefully measured in terms of speed, area and power.

- The execute stage is setup in a way allowing introduction a superscalar way of executing. This means that the execute stage can perform multiple executions of instructions when they use different resources of the execute stage. It is therefore possible that the execute stage calculates an ADD instruction while in parallel executing a MOV instruction.

- Performance can be increased in a relatively easy way by introducing a write buffer in the write back process. With this write buffer it is possible to forward data to the next instruction in the execution scheme. In some cases write back to memories will be done less often, because a next instruction reads from the write buffer and writes back to the original place. E.g. MOV @R0,#data followed by the instruction INC @R0. When a write buffer is being used, one read and one write access to the memory is saved.

- Data sent to the write stage can also be forwarded to the decode/read stage. Mostly data which is sent to the write stage is needed for the next instruction as an argument. This will save a read access to the memory.

- The fetch stage does not use branch prediction. It is not likely that this principle will give a lot of increase in performance due to the high control overhead, but effective implementations of this principle are possible. The advantage is that branch delays will be minimized.

- Sequential buffers instead of using wagging buffers in the fetch stage. Preliminary tests already show that this will improve 17% average performance of the fetch stage and predecode stage.

- The code memory is now accessible through a 4 bytes wide channel, but it is possible that the data memory can also benefit from a parallel access to the memory for instructions like the CALL and RET.

6.5. Final conclusion

The HT80C51-HS now operates at an average speed of 28.3 MIPS in worst case conditions. This is more than three times the performance and the target previously described is reached!
Speeds above 30 MIPS should be achievable when more time is spent on fine tuning and peepholing the microcontroller.
Some bottlenecks and limitations are found in the Handshake Technology design flow such as the handshaking overhead for the memory accesses. They are addressed, and solutions should be developed in a later stage.

As can be concluded from this thesis, high-speed pipelined asynchronous designs are feasible. The Handshake Solutions design environment can deliver high-speed designs. For more complex structures than the 80C51 architecture some other bottlenecks can arise to prevent high-speed designs. The downside of self-timed circuits is; without the use of a centralized clock there is the need for more control logic to assure correct behaviour. The more complex designs are going to be, the more complex the control logic will be. The need exists for smaller control structures, faster assignments to variables and faster channel communications. This could be a complete new study on how to design these new structures.

7. References

[1] Hennessy J.L. & Patterson D.A.
Computer Architecture, A Quantitative Approach, Third Edition
Morgan Kaufmann Publishers 2003

[2] Gageldonk J.S.H.
An Asynchronous Low-Power 80C51 Microcontroller
PhD thesis, Faculty of Mathematics and Computing Science, TUE 1998

[3] 80C51-Based 8-bit Microcontrollers
Data Handbook IC20.
Philips Semiconductors 1997

[4] Peeters A. and de Wit M.
Haste language reference manual. Technical report 2006
http://www.handshakesolutions.com/More_information/Downloads/Index.html

[5] Peeters A.M.G.
Single rail handshake circuits
PhD thesis, Faculty of Mathematics and Computing Science, TUE 1996

[6] Timmermans D.
The design of a Micro-controller, a Transformational Approach
Master thesis, Faculty of Mathematics and Computing Science, TUE 2002

[7] Hoare C.A.R.
Communication Sequential processes
Prentice Hall International Series in Computer Science, 1985

[8] HT80C51 microcontroller
Datasheet on website, 25 Aug 2008
http://www.handshakesolutions.com/products_services/HT-80C51/Index.html

[9] J-H Lee, YH Kim,
Design of a Fast Asynchronous Embedded CISC Microcontroller A8051
IEICE Transaction on Electronics, Vol.,E87-C NO.4, pages 527-534, 2004

[10] H.v. Gageldonk, K.v. Berkel,
An Asynchronous low-power 80C51 Microcontroller
Proc. 4th Int'l Symp. On Advanced Research in Asynchronous Circuits and Systems,
pages 96-107, 1998

[11] A.J. Martin, M. Nystrom, K. Papadantonakis, P. Penzes, P. Prakash, C. Wong, J. Chang, K. Ko, B. Lee, E, Ou, J. Pugh, E. Talvala, J. Tong, A. Tura
The Lutonium: Sub-nanojoule asynchronous 8051 microcontroller
Proc. International Symposium on advanced Research in Asynchronous Circuits and Systems, pages 14-23, IEEE Computer Society Press, May 2003

[12] K. v. Berkel & M. Rem
VLSI programming of asynchronous circuits for low power
Asynchronous Digital Circuit Design, Workshops in Computing, pages 152-210. Springer-Verlag, 1995

[13] 80C51 microcontroller product overview
Datasheet on website, 25 Aug 2008
www.nxp.com/#/homepage/cb=[t=p,p=/50809/45995]|pp=[t=pfp,i=45995]

[14] MCS 251 Architecture overview
Datasheet on website, 25 Aug 2008
www.intel.com/design/mcs51

[15] R8051XC-B 80515-Compatible Microcontroller core
Datasheet on website, 25 Aug 2008
http://www.cast-inc.com/cores/r8051xc/index.shtml

[16] Dolphin Microcontrollers Flip 8050 Cyclone 8-bit microcontroller
Datasheet on website, 25 Aug 2008
http://www.dolphin.fr/flip/logic/8051/logic_8050_cyclone.html

[17] Silicon Labs CT8051T63 product overview
Datasheet on website, 25 Aug 2008
https://www.silabs.com/products/mcu/otp-eprom/Pages/C8051T63x.aspx

[18] Maxim – Product Table DS80Cxxx
Datasheet on website, 25 Aug 2008
http://para.maxim-ic.com/en/search.mvp?fam=hsuc&tree=ucontroller

[19] Atmel Microcontroller 8051 architecture
Datasheet on website, 25 Aug 2008
http://www.atmel.com/products/8051/default.asp

[20] Texas Instruments 8051-based MCUs
Datasheet on website, 25 Aug 2008
http://focus.ti.com/mcu/docs/mcugeneralcontent.tsp?sectionId=98&tabId=1515

[21] DP805X Pipelined High Performance Microcontroller
Datasheet on website, 25 Aug 2008
http://www.dcd.pl/acore.php?idcore=43

[22] J-H Lee, Y.H Kim and K-R Cho
A low-power implementation of asynchronous 8051 employing adaptive pipeline structure
IEEE Transactions on Circuits and Systems, vol. 55 no.7, pages 673-677, 2008

8. List of Tables

9. List of Figures

Appendix

A1 Handshake Solutions

Handshake Solutions is a Line of Business within the Philips Technology Incubator, a program committed to turning promising technologies into successful business entities. As a dynamic organization, Handshake Solutions is dedicated to developing innovative IC design solutions that meet changing market needs. Handshake Solutions is the first to offer a highly disciplined methodology for designing self-timed circuitry that allows commercial exploitation of some of the key benefits of clockless technology – low power consumption, low electromagnetic emissions and low ground bounce. These self-timed designs by Handshake Solutions are easily integrated into complete systems.

A brief history
Handshake Solutions started life as a Philips Research project in 1986. Once the project reached maturity, Handshake Solutions was set up as an internal department to offer the design technology and a full design service to customers within Philips. Over the last ten years Handshake Solutions has worked with Philips Semiconductors (now NXP Semiconductors) to solve difficult yet important issues such as Design-for-Test and prototyping, tailoring the methodology to a commercial IC design environment and bringing numerous successful products to market. Now, with the launch of the Handshake Solutions as a dedicated Line of Business this technology, expertise and services is offered to the wider IC design community.

Philips Technology Incubator
The Philips Technology Incubator creates new businesses based on world-class technologies invented by Philips Corporate Research & Development. It identifies opportunities and helps research teams transform their projects into successful businesses. The formation of business structures around these promising technologies allows a faster take-up by customers and strategic partners. It also creates an excellent opportunity to establish strategic partnerships with leading companies in relevant markets.

Handshake Technology
Handshake Technology is a rigorous design methodology and associated toolset for clockless, self-timed circuits. The familiar global clock used in traditional VLSI design is replaced with a system of request and acknowledgement signals or *handshakes*.

This means that only those parts of a system actively involved in task executions draw power, reducing standby power consumptions to zero and extending battery lifetimes. What's more, individual functions don't have to wait for the next clock pulse, enabling immediate response to exceptional events.

The handshake signalling approach also allows a truly plug-and-play method of integrating (Intellectual Property (IP) from various sources. Because they exhibit no ground bounce, Handshake Technology functions can be quickly and easily combined with clocked logic, analog, RF or memory blocks to meet the exact requirements of your system.

A2 80C51 Instruction set

	0	1	2	3
0	NOP	AJMP (P0) [2B, 2C]	LJMP addr16 [3B, 2C]	RR A
1	JBC bit, rel [3B, 2C]	ACALL (P0) [2B, 2C]	LCALL addr16 [3B, 2C]	RRC A
2	JB bit, rel [3B, 2C]	AJMP (P1) [2B, 2C]	RET [2C]	RL A
3	JNB bit, rel [3B, 2C]	ACALL (P1) [2B, 2C]	RETI [2C]	RLC A
4	JC rel [2B, 2C]	AJMP (P2) [2B, 2C]	ORL dir, A [2B]	ORL dir, #data [3B, 2C]
5	JNC rel [2B, 2C]	ACALL (P2) [2B, 2C]	ANL dir, A [2B]	ANL dir, #data [3B, 2C]
6	JZ rel [2B, 2C]	AJMP (P3) [2B, 2C]	XRL dir, A [2B]	XRL dir, #data [3B, 2C]
7	JNZ rel [2B, 2C]	ACALL (P3) [2B, 2C]	ORL C, bit [2B, 2C]	JMP @A + DPTR [2C]
8	SJMP rel [2B, 2C]	AJMP (P4) [2B, 2C]	ANL C, bit [2B, 2C]	MOVC A, @A + PC [2C]
9	MOV DPTR, #data16 [3B, 2C]	ACALL (P4) [2B, 2C]	MOV bit, C [2B, 2C]	MOVC A, @A + DPTR [2C]
A	ORL C, -bit [2B, 2C]	AJMP (P5) [2B, 2C]	MOV C, bit [2B]	INC DPTR [2C]
B	ANL C, -bit [2B, 2C]	ACALL (P5) [2B, 2C]	CPL bit [2B]	CPL C
C	PUSH dir [2B, 2C]	AJMP (P5) [2B, 2C]	CLR bit [2B]	CLR C
D	POP dir [2B, 2C]	ACALL (P6) [2B, 2C]	SETB bit [2B]	SETB C
E	MOVX A, @DPTR [2C]	AJMP (P7) [2B, 2C]	MOVX A, @R0 [2C]	MOVX A, @R1 [2C]
F	MOVX @DPTR, A [2C]	ACALL (P7) [2B, 2C]	MOVX @R0, A [2C]	MOVX @R1, A [2C]

Note: Key: [2B] = 2 Byte, [3B] = 3 Byte, [2C] = 2 Cycle, [4C] = 4 Cycle, Blank = 1 byte/1 cycle

	4	5	67	8f
0	INC A	INC dir [2B]	INC @R_i	INC R_i
1	DEC A	DEC dir [2B]	DEC @R_i	DEC R_i
2	ADD A, #data [2B]	ADD A, dir [2B]	ADD A, @R_i	ADD A, R_i
3	ADDC A, #data [2B]	ADDC A, dir [2B]	ADDC A, @R_i	ADDC A, R_i
4	ORL A, #data [2B]	ORL A, dir [2B]	ORL A, @R_i	ORL A, R_i
5	ANL A, #data [2B]	ANL A, dir [2B]	ANL A, @R_i	ANL A, R_i
6	XRL A, #data [2B]	XRL A, dir [2B]	XRL A, @R_i	XRL A, R_i
7	MOV A, #data [2B]	MOV dir, #data [3B, 2C]	MOV @R_i, #data [2B]	MOV R_i, #data [2B]
8	DIV AB [4C]	MOV dir, dir [3B, 2C]	MOV dir, @R_i [2B, 2C]	MOV dir, R_i [2B, 2C]
9	SUBB A, #data [2B]	SUBB A, dir [2B]	SUBB A, @R_i	SUBB A, R_i
A	MUL AB [4C]		MOV @R_i, dir [2B, 2C]	MOV R_i, dir [2B, 2C]
B	CJNE A, #data, rel [3B, 2C]	CJNE A, dir, rel [3B, 2C]	CJNE @R_i, #data, rel [3B, 2C]	CJNE R_i, #data, rel [3B, 2C]
C	SWAP A	XCH A, dir [2B]	XCH A, @R_i	XCH A, R_i
D	DA A	DJNZ dir, rel [3B, 2C]	XCHD A, @R_i	DJNZ R_i, rel [2B,2C]
E	CLR A	MOV A, dir [2B]	MOV A, @R_i	MOV A, R_i
F	CPL A	MOV dir, A [2B]	MOV @R_i, A	MOV R_i, A

Note: Key: [2B] = 2 Byte, [3B] = 3 Byte, [2C] = 2 Cycle, [4C] = 4 Cycle, Blank = 1 byte/1 cycle

A3 Configuration of the HT80C51

All the HT80C51s are configured the same way, used the same tools, testbenches and technologies to guarantee the most reliable comparisons between the different microcontrollers.
The library used to map every microcontroller on is CMOS 0.14 µm technology. Third party tools are all tools by Cadence like RTL compiler. Every HT80C51 measurement is done with memories access speed of 5 ns and the RTL compiler has optimized twice (double run). All measurements are in worst case scenario. Power is measured by Diesel.

The speed and area numbers of the HT80C51s are without peripherals (timers, interrupt controller, CPU extensions (MX), debug units (OCI, memory map). No scan-chain is inserted.

The following testbenches are used to validate the HT80C51-SU and HT80C51-HS cores:

- ccore_0.asm
- ccore_1.asm
- ccore_2.asm
- ccore_3.asm
- ccore_4.asm
- testbench_short.asm
- main65.asm
- instr.asm
- Some own written testbenches

A4 Instruction execution time and memory specification HT80C51

Mnemonic	Opcode	Execution cycles standard 80C51	Instruction bytes (ROM)	RAM Read	RAM Write
ADD A, Rn	28-2F	12	1	1	
ADD A, direct	25	12	2	1	
ADD A, @Ri	26-27	12	1	2	
ADD A, #data	24	12	2		
ADDC A, Rn	38-3F	12	1	1	
ADDC A, direct	35	12	2	1	
ADDC A, @Ri	36-37	12	1	2	
ADDC A, #data	34	12	2		
SUB A, Rn	98-9F	12	1	1	
SUB A, direct	95	12	2	1	
SUB A, @Ri	96-97	12	1	2	
SUB A, #data	94	12	2		
INC A	4	12	1		
INC Rn	8-F	12	1	1	1
INC direct	5	12	2	1	1
INC @Ri	6-7	12	1	2	1
DEC A	14	12	1		
DEC Rn	18-1F	12	1	1	1
DEC direct	15	12	2	1	1
DEC @Ri	16-17	12	1	2	1
INC DPTR	A3	24	1		
MUL AB	A4	48	1		
DIV AB	84	48	1		
DA A	D4	12	1		

[Table 20] Arithmetic operation execution time and memory specification HT80C51

Mnemonic	Opcode	Execution cycles standard 80C51	Instruction bytes (ROM)	RAM Read	RAM Write
ANL A, Rn	58-5F	12	1	1	
ANL A, direct	55	12	2	1	
ANL A, @Ri	56-57	12	1	2	
ANL A, #data	54	12	2		
ANL direct, A	52	12	2	1	1
ANL direct, #data	53	24	3	1	1
ORL A, Rn	48-4F	12	1	1	
ORL A, direct	45	12	2	1	
ORL A, @Ri	46-47	12	1	2	
ORL A, #data	44	12	2		
ORL direct, A	42	12	2	1	1
ORL direct, #data	43	24	3	1	1
XRL A, Rn	68-6F	12	1	1	
XRL A, direct	65	12	2	1	
XRL A, @Ri	66-67	12	1	2	
XRL A, #data	64	12	2		
XRL direct, A	62	12	2	1	1
XRL direct, #data	63	24	3	1	1
CLR A	E4	12	1		
CPL A	F4	12	1		
RL A	23	12	1		
RLC A	33	12	1		
RR A	3	12	1		
RRC A	13	12	1		
SWAP A	C4	12	1		

[Table 21] Logical operation execution time and memory specification HT80C51

Mnemonic	Opcode	Execution cycles standard 80C51	Instruction bytes (ROM)	RAM Read	RAM Write
MOV A, Rn	E8-EF	12	1	1	
MOV A, direct	E5	12	2	1	
MOV A, @Ri	E6-E7	12	1	2	
MOV A, #data	74	12	2		
MOV Rn, A	F8-FF	12	1		1
MOV Rn, direct	A8-AF	24	2	1	1
MOV Rn, #data	78-7F	12	2		1
MOV direct, A	F5	12	2		1
MOV direct, Rn	88-8F	24	2	1	1
MOV direct, direct	85	24	3	1	1
MOV direct, @Ri	86-87	24	2	2	1
MOV direct, #data	75	24	3		1
MOV @Ri, A	F6-F7	12	1	1	1
MOV @Ri, direct	A6-A7	24	2	2	1
MOV @Ri, #data	76-77	12	2	1	1
MOV DPTR, #data16	90	24	3		
MOVC A, @A+DPTR	93	24	1		
MOVC A, @A+PC	83	24	1		
MOVX A, @Ri	E2-E3	24	1	2	
MOVX A, @DPTR	E0	24	1	1	
MOVX @Ri, A	F2-F3	24	1	1	1
MOVX @DPTR, A	F0	24	1	1	1
PUSH direct	C0	24	2	1	1
POP direct	D0	24	2	1	1
XCH A, Rn	C8	12	1	1	1
XCH A, direct	C5	12	2	1	1
XCH A, @Ri	C6-C7	12	1	2	1
XCHD A, @Ri	D6-D7	12	1	2	1

[Table 22] Data transfer execution time and memory specification HT80C51

Mnemonic	Opcode	Execution cycles standard 80C51	Instruction bytes (ROM)	RAM Read	RAM Write
CLR C	C3	12	1		
CLR bit	C2	12	2	1	1
SETB C	D3	12	1		
SETB bit	D2	12	2	1	1
CPL C	B3	12	1		
CPL bit	B2	12	2	1	1
ANL C, bit	82	24	2	1	
ANL C, -bit	B0	24	2	1	
ORL C, bit	72	24	2	1	
ORL C, -bit	A0	24	2	1	
MOV C, bit	A2	12	2	1	
MOV C, -bit	92	24	2		1

[Table 23] Boolean variable manipulation execution time and memory specification HT80C51

Mnemonic	Opcode	Execution cycles standard 80C51	Instruction bytes (ROM)	RAM Read	RAM Write
JC rel	40	24	2		
JNC rel	50	24	2		
JB bit, rel	20	24	3	1	
JNB bit, rel	30	24	3	1	
JBC, bit, rel	10	24	3	1	1
ACALL P0-P7	11-F1	24	2		2
LCALL addr16	12	24	3		2
RET	22	24	1	3	
RETI	32	24	1	3	
AJMP P0-P7	01-E1	24	2		2
LJMP addr16	2	24	3		
SJMP rel	80	24	2		
JMP @A+DPTR	73	24	1		
JZ rel	60	24	2		
JNZ rel	70	24	2	1	
CJNE A, direct, rel	B5	24	3	1	
CJNE A, #data, rel	B4	24	3		
CJNE Rn, #data, rel	B8-BF	24	3	1	
CJNE @Ri, #data, rel	B6-B7	24	3	2	
DJNZ Rn, rel	D8-DF	24	2	1	1
DJNZ direct, rel	D5	24	2	1	1
NOP	0	12	1		

[Table 24] Program branching execution time and memory specification HT80C51

A5 Instruction groups specified for HT80C51-LC and HT80C51-SU

HT80C51 instruction grouped to 18 categories.

Instruction group	Instruction number (hex)
1	00 – 03 – 04 – 13 – 14 – 23 – 33 – 84 – a3 – a4 – b3 – c3 – c4 – d3 – d4 – e4 – f4
2	f0 – f8
3	28 – 38 – 48 – 58 – 68 – 98 – e0 – e8
4	08 – 18 – c8 – f3 – f7
5	27 – 37 – 47 – 57 – 67 – 97 – e3 – e7
6	07 – 17 – c7 – d7
7	22 – 32
8	21 – 24 – 34 – 44 – 54 – 64 – 74 – 94 – 40 – 50 – 60 – 70 –80
9	78 – f5
10	31
11	25 – 35 – 45 – 55 – 65 – 95 – e5 – 72 – 82 – a0 – a2 –b0
12	05 – 15 – 42 – 52 – 62 – 92 – 88 – a8 – b2 – c2 – c5 – d2 – d8 – c0 – d0 – 77 – 87 – a7
13	02 – 90 – b4
14	75
15	12
16	20 – 30 – b5 – b8
17	10 – 43 – 53 – 63 – 85 – d5
18	b7

Note: Instructions x8h to xFh are taken together. The last three bits of the instruction code specify a register (0..7) in a register bank. The same holds for instructions x6h and x7h that involve the instructions using indirect addressing: the last bit represents the register (0 or 1) that contains the address of the register to be operated on.

[Table 25] HT80C51-LC and HT80C51-SU instruction groups

A6 Instruction groups specified for HT80C51-HS

Instruction group	Instruction number (hex)
1	00 – 01 – 02 – 03 – 04 – 08 – 13 – 14 – 18 – 23 – 24 – 28 – 33 – 34 – 38 – 40 – 44 – 48 – 50 – 54 – 58 – 60 – 64 – 68 – 70 – 80 – 74 – 78 – 84 – 90 – 94 – 98 – a3 – a4 – b3 – b4 – b8 – c3 – c4 – c8 – d3 – d4 – d8 – e4 – e8 – f4 – f8
2	75 – 76 – 88 – 92 – f0 – f2 – f3 – f5 – f6
3	10 – 20 – 30 – 25 – 26 – 35 – 36 – 45 – 46 – 55 – 56 – 65 – 66 – 72 – 73 – 82 – 83 – 93 – 95 – 96 – a0 – a2 – a8 – b0 – b5 – b6 – e0 – e2 – e3 – e5 – e6
4	05 – 06 – 15 – 16 – 42 – 43 – 52 – 53 – 62 – 63 – 85 – 86 – a6 – b2 – c0 – c2 – c5 – c6 – d0 – d2 – d5 – d6
5	11 – 12 – 31 – 51 – 71 – 91 – b1 – d1 – f1
6	22 – 32

Note: Instructions x8h to xFh are taken together. The last three bits of the instruction code specify a register (0..7) in a register bank. The same holds for instructions x6h and x7h that involve the instructions using indirect addressing: the last bit represents the register (0 or 1) that contains the address of the register to be operated on.

[Table 26] HT80C51-HS instruction groups

A7 Document History

Document History			
Date	**Author**	**Version-No**	**Change Report**
04-12-07	T.J.H. van Hoek	0.1	Initial draft
09-12-08	T.J.H. van Hoek	1.0	Final version Master Thesis