

# **An Asynchronous Java Processor for Smart Card**

YU Chun-pong

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Philosophy

in

Electronic Engineering

©The Chinese University of Hong Kong

July 2003

The Chinese University of Hong Kong holds the copyright of this thesis. Any person(s) intending to use a part of whole of the materials in the thesis in a proposed publication must seek copyright release from the Dean of the Graduate School.



**Abstract of this thesis entitled:**

**An Asynchronous Java Processor for Smart Card**

**Submitted by YU Chun-pong**

**for the degree of Master of Philosophy in Electronic Engineering**

**at The Chinese University of Hong Kong in July 2003**

This thesis presents the design of a low power asynchronous java processor for contactless smart card. The Java Card specifications of Sun Microsystems enable Java technology to run on smart cards. By using a compiler, java source codes can be compiled to java bytecodes. The java processor is a microprocessor that can execute the java bytecodes.

The proposed java processor can directly execute the java bytecodes in a subset of the instruction set defined in the Java Card Virtual Machine specification. The remaining java bytecodes are handled by software routines. Also, we intend to use asynchronous circuit design technique to reduce the power consumption of the java processor.

In order to simplify the design, we have considered the processor as a fixed-length instruction set processor executing instructions with 8-bit opcodes and 16-bit operands.

In this design, we use four-phase bundled-data protocol for normally opaque latch controller as the handshaking protocol in the asynchronous pipeline. In this thesis, we also introduce some asynchronous control elements for this type of protocol.

## 摘要

本論文介紹了一個適用於非接觸型智能卡的低功率異步 Java 處理器設計，昇陽電腦公司(Sun Microsystems)的 Java 智能卡規格使 Java 技術能夠於智能卡中運作。利用編譯器，Java 原始碼能夠被編譯成 Java bytecodes，而 java 處理器就是能夠執行 Java bytecodes 的微處理器。

我們建議的 Java 處理器能夠直接執行定義於 Java Card Virtual Machine specification 的指令集中的一個子集的 Java bytecodes，其餘的 Java bytecodes 則由軟件處理。另外，我們打算利用異步電路設計技術以減低 Java 處理器的功率消耗。

爲了簡化設計，我們把那處理器設計爲一個能執行由 8-bit opcode 與 16-bit operand 組成的指令的 fixed-length instruction set processor。

在這個設計中，我們應用了適用於 normally opaque latch controller 的 four-phase bundle-data protocol 作爲異步管線的聯絡協定。而在這論文中，我們亦會介紹一些適用於此協定的異步控制原件。

## **Acknowledgements**

I would like to take this opportunity to thank my supervisor, Professor Choy Chiu-Sing for his full support on my work throughout the period of study. Moreover, I would like to thank all my peers and colleagues in ASIC/VLSI laboratory. They include Chan Shek-Hang, Chan Wing-Kin, Ho Kin-Pui, Han Wei, Leung Pak-Keung, Yeung Wing-Ki, Cheng Wang-Chi, Hon Kwok-Wai, Chan Chi-Hong, Chan Pak-Kee, Cheng Wang-Tung, Kwok Yan-Lun, Shen Jun-Hua, Tang Siu-Kei and our laboratory technician Mr. Yeung Wing-Yee. Also, I would like to thank my family members and friends for their continuous support and encouragement.

## Table of contents

<b>Abstract of this thesis entitled:</b> .....	<b>i</b>
<b>摘要</b> .....	<b>iii</b>
<b>Acknowledgements</b> .....	<b>iv</b>
<b>Table of contents</b> .....	<b>v</b>
<b>List of Tables</b> .....	<b>vi</b>
<b>List of Figures</b> .....	<b>vii</b>
<b>Chapter 1</b> .....	<b>1</b>
Introduction	
1.1 Asynchronous design .....	1
1.2 Java processor for contactless smart card [3] .....	2
1.3 Motivation.....	3
<b>Chapter 2</b> .....	<b>5</b>
Asynchronous circuit design techniques	
2.1 Overview .....	5
2.2 Handshake protocol .....	5
2.3 Asynchronous pipeline.....	7
2.4 Asynchronous control elements .....	9
<b>Chapter 3</b> .....	<b>15</b>
Asynchronous Java Processor	
3.1 Instruction Set .....	15
3.2 Architecture of the java processor .....	17
3.3 Basic building blocks of the java processor.....	22
3.4 Token flow .....	32
<b>Chapter 4</b> .....	<b>37</b>
Results and Discussion	
4.1 Simulation Results of test programs .....	37
4.2 Experimental result .....	41
4.3 Future work.....	42
<b>Chapter 5</b> .....	<b>45</b>
Conclusion	
<b>Appendix</b> .....	<b>47</b>
Chip micrograph for the java processor core .....	47
Pin assignment of the java processor .....	48
Schematic of the java processor.....	52
Schematic of the decoder.....	54
Schematic of the Stage2 of the java processor.....	55
Schematic of the stack .....	56
Schematic of the block of the local variables .....	57
Schematic of the 16-bit self-timed adder .....	58
The schematic and the layout of the memory cell .....	59
<b>Reference</b> .....	<b>60</b>

List of Tables

Table 3-1 Instructions in the subset..... 16

Table 3-2 hardware features ..... 17

Table 3-3 Paths for each instruction..... 33

Table 4-1 Simulation Results..... 40

Table 4-2 Experimental Results ..... 41

Table 4-3 Result comparison..... 41

## List of Figures

Figure 2-1 (a) bundled data channel (b) 2-phase bundled-data protocol (c) 4-phase bundled-data protocol .....	5
Figure 2-2 A delay-insensitive channel using the 4-phase dual-rail protocol.....	7
Figure 2-3 (a) asynchronous pipeline for simple logic (b) asynchronous pipeline for self-timed logic.....	7
Figure 2-4 The normally opaque latch controller: (a) symbol, (b) STG and (c) speed independent implementation [9] .....	7
Figure 2-5 Delay element used in [9] .....	8
Figure 2-6 (a) The proposed Delay element (b) Possible implementation .....	8
Figure 2-7 (a) fork I [2] (b) fork II (c) fork III.....	10
Figure 2-8 (a) Join [2] (b) Merge [2].....	11
Figure 2-9 Three stage synchronous pipeline .....	11
Figure 2-10 Three stage asynchronous pipeline .....	12
Figure 2-11 Control circuit of the multiplexer .....	13
Figure 3-1 Execution of instructions: (a) iload index (b) istore index (c) iadd ...	15
Figure 3-2 Block diagram of asynchronous java processor (black boxes represent latches) .....	19
Figure 3-3 Fork of request signals generated by the Decoder.....	20
Figure 3-4 Fork of request signals generated by the Stack.....	21
Figure 3-5 Schematic of block of local variables (black boxes represent latches) ..	22
Figure 3-6 Request signals transmitted by the latch controller of the Latch_RW .....	23
Figure 3-7 Stack (black boxes represent latches).....	24
Figure 3-8 (a) Ram cell (b) Detection circuit (c) A column of the RAM.....	27
Figure 3-9 Fetch/decode unit (black boxes represent latches).....	28
Figure 3-10 Request signals form the latch controller of the operand latch.....	29
Figure 3-11 (a) Basic adder cell (b) 4-bit adder (c) 16-bit adder.....	30
Figure 3-12 Paths for transmitting control bits and data bits (path 1, path 2 and path 3 are the paths for transmitting control bits) .....	32
Figure 3-13 Execution of 'iload index': (a) Step 1 (b) Step 2 .....	33
Figure 3-14 Execution of 'sipush const': (a) Step 1 (b) Step 2 .....	34
Figure 3-15 Execution of 'iadd': (a) Step 1 (b) Step 2 (c) Step 3 .....	34
Figure 3-16 Execution of 'if_scmpeq_w offset': (a) Step 1 (b) step2 (c) Step3 ..	35
Figure 4-1 modified architecture of the java processor .....	43

## **Chapter 1**

### **Introduction**

#### **1.1 Asynchronous design**

Nowadays, most fabricated integrated circuits are synchronous circuits. In a synchronous system, all the logics are driven by a global clock signal. Such integrated circuits are easy to design and test since the circuit can only operate after the positive/negative edge of the clock signal. Also, there are many sophisticated design tools to help designing a synchronous chipset.

However, in a synchronous system, since the clock signal and its driven logic will continue to operate even if there is no useful work to perform, power is wasted for unnecessary transistor switching.

In order to reduce the unessential power loss, asynchronous circuit design technique is introduced. As in an asynchronous system, most transitions are useful work since all the logics are data driven. [1] Also, other advantages of asynchronous circuits include increasing the operating speed, alleviating clock slew problem and reducing the emission of electro-magnetic noise. [2]

## **1.2 Java processor for contactless smart card [3]**

A smart card is a device that has both processing power and memory and it is packaged in the format defined by the International Standards Organization (ISO). There are two types of smart card including memory smart card and intelligent smart card.

The memory smart card is a purely memory storage card and it does not consist of a microprocessor. Instead, some non-processor chipset that have hard wired logic is used to control the access security. Once linked to the outside world, they are powered, clocked and addressed totally under control of the outside world.

The intelligent smart card contains a memory module and a Central Processing Unit (CPU) with the abilities to store and secure information, the power to make decisions and read/write capabilities. Since the microprocessor chipset for storing and manipulating data is embedded, a smart operating system can be built. Due to the intelligence of the microprocessor, it can afford greater security. There are many applications of smart card in various areas now. Currently, smart cards can be used for payment of telephone calls, payment of parking and tolls, storage of identification and medical records and access to some restricted areas.

Also, smart card can be classified as contact smart card and contactless smart. There are some metallic contact pads on the surface of the contact smart card. The contacts

link the logics in the smart card with the read/write unit (Smart Card Reader) to enable communication between them. Instead, for the contactless smart card, electrical coupling is used for such communication. So, when the contactless smart card is placed in close proximity to a reader, about less than 3 centimeters, input modulated signal and power signal can be received by the logics in the contactless smart card.

The Java Card specifications of Sun Microsystems enable Java technology to run on smart cards. There are several benefits of the Java Card technology, such as interoperable, Secure, multi-Application Capable and compatible with Existing Standards. Java programming language is a high level language. By using a compiler, java source code can be compiled to java bytecode. The java processor is a microprocessor that can execute the java bytecode.

### **1.3 Motivation**

Java Card technology enables programs written in the Java programming language to run in smart cards. From software perspective, Java virtual machine defines a stack-based processor for implementing java bytecodes. To implement it on conventional 8-bit processors, such as Intel's 8051 and Motorola's 6805, a software virtual machine is used to emulate its architecture. This not only occupies the limited memories on the smart card but also slows down the operating speed of the java card. Also, it is possible to design a microprocessor that can execute these bytecodes directly in order to increase the operating performance. This type of synchronous java

processor for contact smart card is proposed in [4]. It consists of six stages: Instruction Fetch, Instruction Decode, Operand Access, Execution, Memory Access and Write back.

Since there are only a few proposed designs of such type of java processor and even less in the contactless smart card context, a novel asynchronous java processor is presented in this thesis.

Asynchronous circuit design technique is used for reducing the power consumption of the java processor core since low power consumption is a very important constraint for the application of contactless smart card.

## Chapter 2

### Asynchronous circuit design techniques

#### 2.1 Overview

The operation of synchronous circuit is controlled by a clock signal. In an asynchronous circuit, the clock signal is replaced by some form of handshaking. There are two types of handshaking protocol including 2-phase protocol and 4-phase protocol. For each protocol, there are different types of handshake cell for controlling the operation of the asynchronous pipeline. [2]

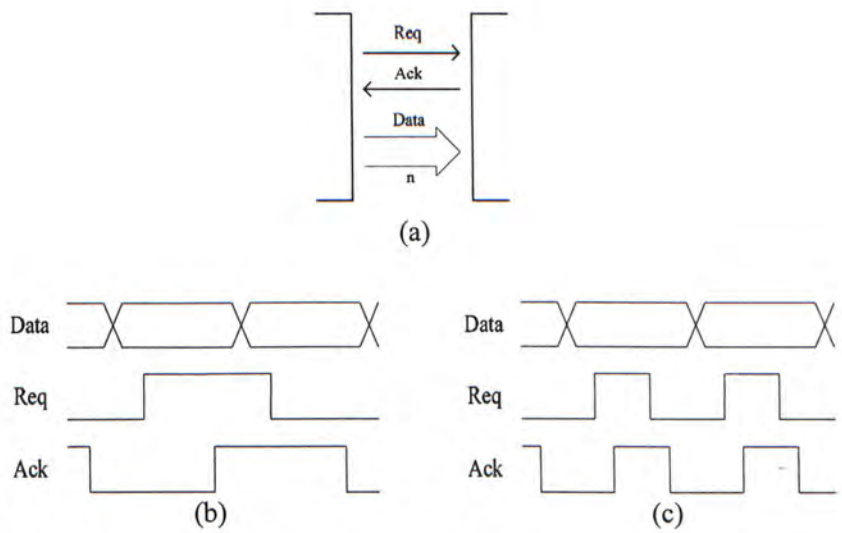


Figure 2-1 (a) bundled data channel (b) 2-phase bundled-data protocol (c) 4-phase bundled-data protocol

#### 2.2 Handshake protocol

The most popular handshake protocols are 2-phase bundled-data protocol, 4-phase bundled-data protocol and 4-phase dual-rail protocol. The bundled data channel is

shown in figure 2-1(a). A request signal (Req) and an acknowledge signal (Ack) are bundled with the data signal to control the communication between the sender and receiver.

For the 2-phase bundled-data protocol, the information on the request and acknowledge wires is encoded as signal transition as shown in figure 2-1(b). Both a 0 to 1 transition and a 1 to 0 transition represent “signal events”. When the data from the sender is valid, the request signal is changed from low to high. When the data is received by the receiver, the acknowledge signal is changed from low to high or vice versa. The 4-phase bundled-data protocol is shown in figure 2-1(b). The term 4-phase refers to the number of communication actions: (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 and (4) the receiver acknowledges this by taking acknowledge low. [2] Ideally, the 2-phase bundled-data protocol is more effective than 4-phase bundled-data protocol since the 1 to 0 transition of the 4-phase bundled-data protocol seems to increase the complexity of the interface protocol and costs unnecessary time and energy. However, the control elements used in the 2-phase bundled-data protocol such as toggle, select and call [5] [6] are more complicated than that of the 4-phase bundled-data protocol. Also, in order to detect the completion signal of some self-timed circuit such as self-timed memory and self-timed adder [7], the 2-phase bundled-data protocol does not work properly. In this situation, it has to be converted to 4-phase bundled-data protocol by a 2-phase to 4-phase converter. [8] We choose the 4-phase bundled-data protocol as the interface protocol for the proposed processor design since there are a lot of control elements and self-timed circuits utilized in the processor core.

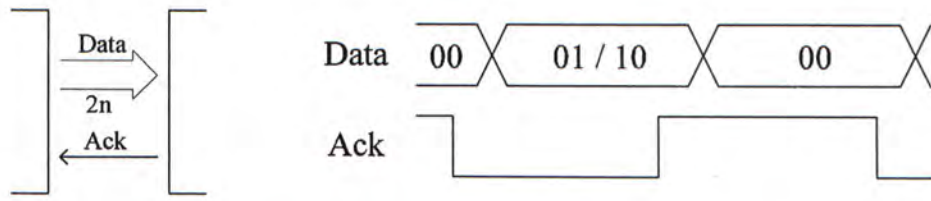


Figure 2-2 A delay-insensitive channel using the 4-phase dual-rail protocol

The 4-phase dual-rail protocol encodes the request signal into the data signals using two wires per bit of information as shown in figure 2-2. It is not selected for the interface protocol used in the processor because the number of transistor switching is quite large for this protocol (This will increase the power consumption).

2.3 Asynchronous pipeline

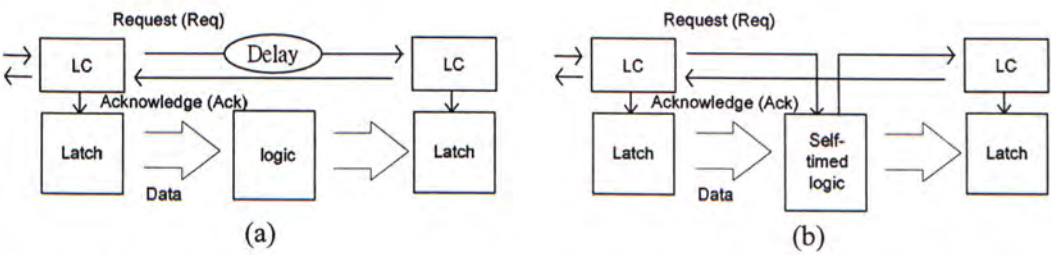


Figure 2-3 (a) asynchronous pipeline for simple logic (b) asynchronous pipeline for self-timed logic

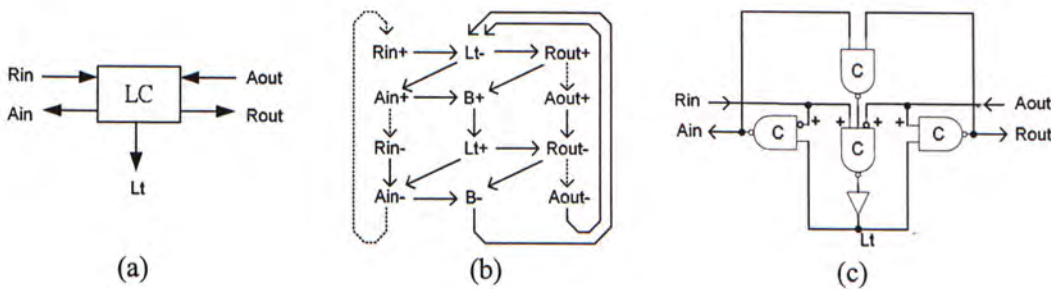


Figure 2-4 The normally opaque latch controller: (a) symbol, (b) STG and (c) speed independent implementation [9]

We use 4-phase bundled-data protocol in the processor design. The asynchronous pipeline for s imple l ogic c ircuit and the asynchronous p ipeline for self-timed l ogic

circuit are shown in figure 2-3(a) and figure 2-3(b) respectively. All the latches are manipulated by the latch controllers (LC). Since the normally opaque latch controller is used, all the latches will be closed after reset. When the latch controller receives a request signal, the latch controlled by it will be opened and then closed to store the input data. The symbol, STG and speed independent implementation of the normally opaque latch controller are shown in figure 2-4(a), 2-4(b) and 2-4(c). [9] Before the rising edge of Rout, the data is valid. The data may not be valid after the falling edge of the Aout.

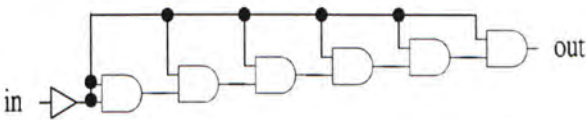


Figure 2-5 Delay element used in [9]

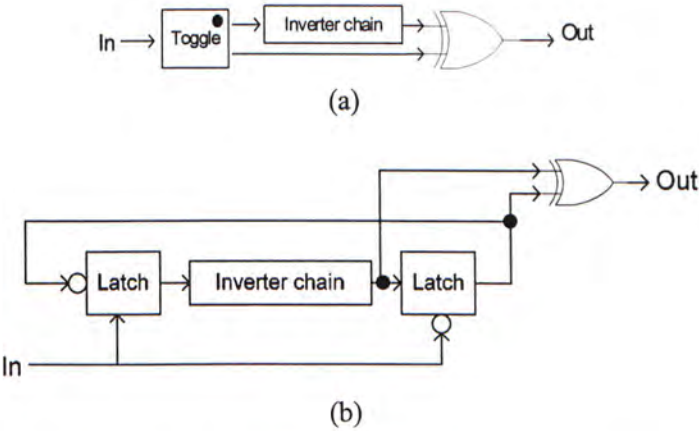


Figure 2-6 (a) The proposed Delay element (b) Possible implementation

The delay element shown in figure 2-3(a) can provide a constant low-to-high propagation delay that matches the worst case latency of the logic circuit. It is used to generate a completion signal for the receiver. For this type of protocol, no delay is needed to provide for the high-to-low propagation in the delay element. The delay element used in [9] as shown in figure 2-5 can provide a fast high-to-low propagation

delay but it wastes much power in this type of propagation when the delay is very large since the energy used for this type of propagation is about directly proportional to the required delay. In order to reduce the power consumption, we proposed a new delay element as shown in figure 2-6(a). When the input signal changes from low to high, the signal will propagate in the upper path through the inverter chain. When the input signal changes from high to low, the signal will propagate in the lower path. For the high-to-low propagation in the delay element, only a small amount of constant energy is consumed. So, much power can be reduced when the delay is large. A possible implementation is shown in Figure 2-6(b) and the toggle is replaced by two level-sensitive latches and two NOT gates. We have assumed that the delay of the inverter chain is large enough to avoid the race hazard when both latches are transparent and the signal can transmit through them at the same time. Simulation result (0.35um CMOS process) shows that more than 40% of power dissipation on the delay element is reduced using the circuit shown in Figure 2-6(b) when the matched delay is more than 3ns.

Instead of the delay element, the completion signal is generated by a self-timed circuit in the asynchronous pipeline shown in figure 2-3(b). Self-timed adders and self-timed memories are used in the processor design.

## **2.4 Asynchronous control elements**

There are three types of control elements including fork, join and merge used in the processor core. In order to simplify the explanation, we only consider the control elements for manipulating two request signals or two acknowledge signals in this section.

The function of the fork element is to distribute the request signal(s) to receiver(s) and collect the corresponding acknowledge signal(s) from the receiver(s). We call the fork element in [2] as shown in figure 2-7(a) fork I. The request signal (Rin) from sender will always pass to both receivers by the fork I.

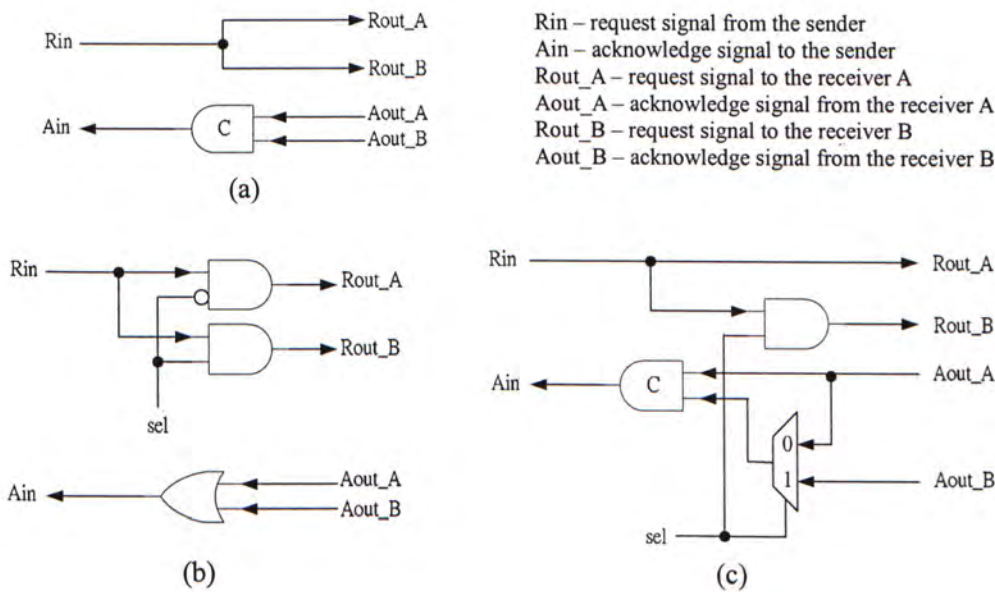


Figure 2-7 (a) fork I [2] (b) fork II (c) fork III

For the Fork element II shown in Figure 2-7(b), either one of the receiver A and receiver B will receive the request signal controlled by the signal sel generated by the sender. If sel is 0, a request signal (Rout\_A) is transmitted to the receiver A. If sel is 1, a request signal (Rout\_B) is transmitted to the receiver B.

The third fork element used in the processor core is fork III shown in figure 2-7 (c). The request signal from the sender will always pass to the receiver A. Also, receiver B will receive a request signal controlled by the signal sel generated by the sender. If sel is 0, no request signal is transmitted to the receiver B. If sel is 1, a request signal

(Rout\_B) is transmitted to the receiver B. Also, similar to other data signals, the input signal sel for the fork II/fork III is valid before the rising edge of the rin signal.

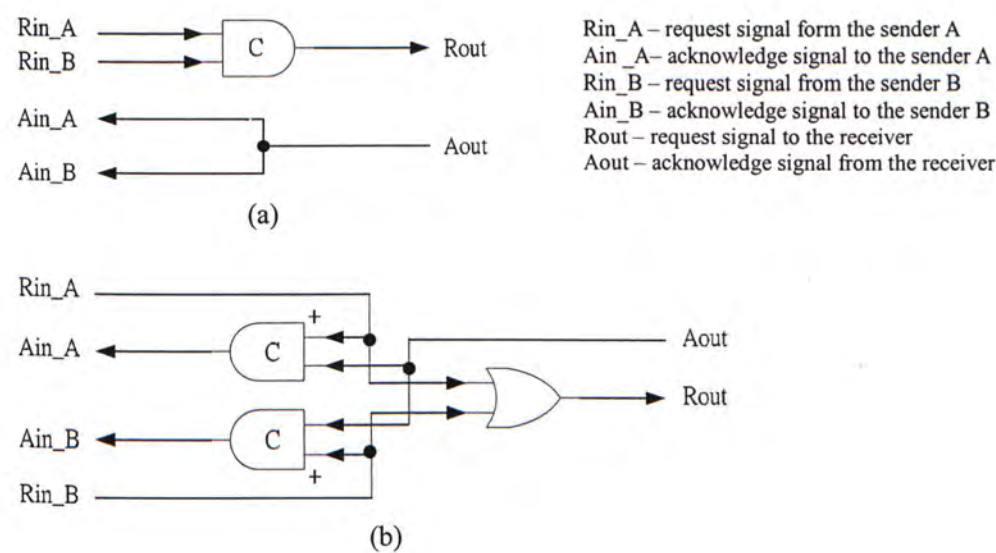


Figure 2-8 (a) Join [2] (b) Merge [2]

The 4-phase join [2]and the 4-phase merge [2] are shown in figure 2-8(a) and 2-8(b) respectively. For the 4-phase join, a request signal will be sent to the receiver if both the sender A and sender B have transmitted request signals (Rin\_A and Rin\_B) to it. For the 4-phase merge, a request signal will be sent to the receiver if one of the sender A and sender B has transmitted a request signal (Rin\_A or Rin\_B) to it.

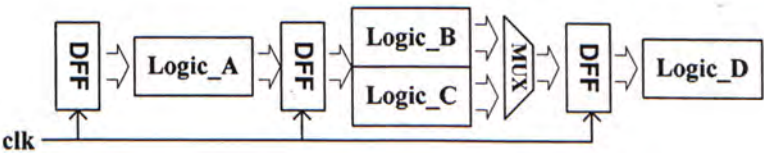


Figure 2-9 Three stage synchronous pipeline

To see how they work, we consider a simple three-stage synchronous pipeline shown in Figure 2-9. All the Flip-flops DFF are controlled by a clock signal clk. logic\_A,

logic\_B, logic\_C and logic\_D are combinational circuits. In this situation, only one of the results of logic\_B and logic\_C is needed for the operation of logic\_D in the third stage and the control bit for the MUX is generated by logic\_A in the previous cycle. For the synchronous pipeline, both logic\_B and logic\_C have to operate since they are controlled by a single clock signal.

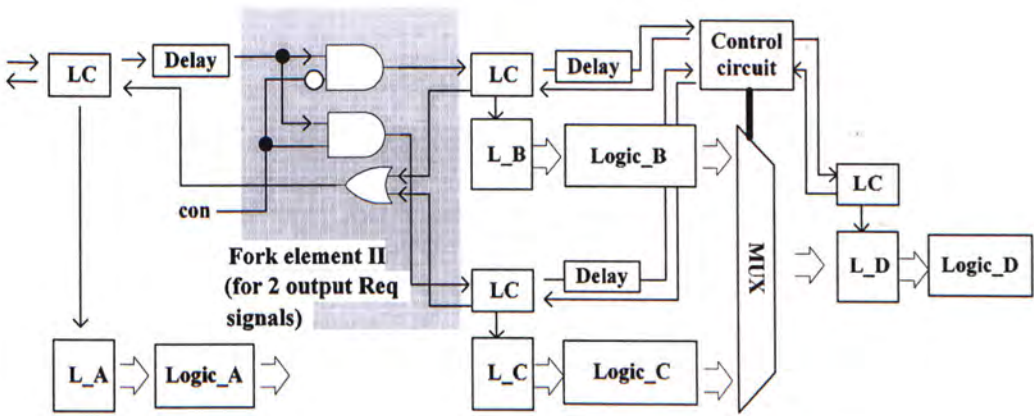


Figure 2-10 Three stage asynchronous pipeline

The asynchronous pipeline shown in Figure 2-10 can perform the same operation of the synchronous pipeline shown in figure 2-9. However, for this asynchronous pipeline, only one of logic\_B and logic\_C will operate when necessary. The normally opaque latch controller LC controls the operation of the latches L\_A, L\_B, L\_C or L\_D. The control bit con generated by logic\_A is used to select one of the operations of logic\_B and logic\_C so that the request signal can transmit through the fork element II to one of the latch controllers LC. When there are more than two latch controllers in the second stage, more AND gates, more control bits and the OR gate with more input pins will be used for the fork element II.

There are two functions of the control circuit of the multiplexer shown in 2-10. The first function is to transmit a request signal to the LC of L\_D when it has received one of the request signals from LC of L\_B and LC of L\_C. So, this control circuit can be implemented by using the merge element shown in 2-8(b). The second function is to manipulate the control bits of the multiplexers for selecting one the data buses from logic\_B and logic\_C.

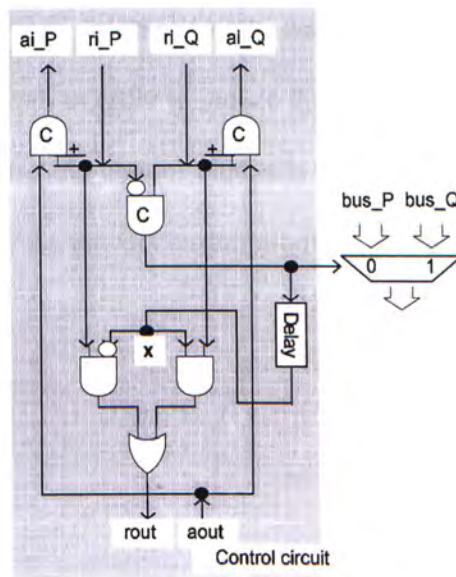


Figure 2-11 Control circuit of the multiplexer

According to the two functions, the control circuit of the multiplexer shown in 2-11 is designed. It is also used for the control circuits for multiplexing two buses used in the java processor.  $ri\_P$  and  $ri\_Q$  are the request signals for bus P and bus Q respectively and  $ai\_P$  and  $ai\_Q$  are the acknowledge signals for bus P and bus Q respectively. When  $ri\_P$  changes from low to high, the control bit of the multiplexer is set to zero in order to select bus P. Similarly, when  $ri\_Q$  changes from low to high, the control bit of the multiplexer is set to one in order to select bus Q. The selection of the previous access of the multiplexer is stored in node X. If the current selection is the same as the

previous selection, the request input signal will be directly transmitted to the output. Otherwise, it will be transmitted through the delay element and change the value at node X.

By using the handshake signals controlled by the control elements, we can control the operations of the logic blocks in order to reduce the power consumption. However, the larger the number of control elements we used in the design, the more complicated is the handshake signal path. In other words, we increase the power consumption of the handshaking signal path in order to reduce the unnecessary operations of the logic blocks. So, before we use the control elements to distribute the request signals, we need to evaluate whether the power consumption can be reduced.



istore index	A value is popped from the stack. A local variable is set to this value (figure 3-1(b))
iadd	Two values are popped from the stack. The result of addition of this two values is pushed onto the stack (figure 3-1(c))
isub	Two values are popped from the stack. The result of subtraction of this two values is pushed onto the stack
ishl	Two values are popped from the stack. The result calculated by left shifting one of the value by n bit(s) (n is the value of the lower two bits of the other value) is pushed onto the stack
ishr	Two values are popped from the stack. The result calculated by right shifting one of the value by n bit(s) (n is the value of the lower two bits of the other value) is pushed onto the stack
iand	Two values are popped from the stack. The result calculated by the bitwise AND of this two values is pushed onto the stack
ior	Two values are popped from the stack. The result calculated by the bitwise OR of this two values is pushed onto the stack
ixor	Two values are popped from the stack. The result calculated by the bitwise XOR of this two values is pushed onto the stack
if_scmpcq_w offset	Two values are popped from the stack. Branch if the two values are the same
if_scmlt_w offset	Value1 and Value2 are popped from the stack one by one. Branch if Value2 is less than Value1
if_scmpge_w offset	Value1 and Value2 are popped from the stack one by one. Branch if Value2 is greater than or equal to Value1
goto offset	Branch always
jsr offset.	Jump subroutine
ret	Return from subroutine. The return value is popped from the stack
nop	No operation
s2p	A value is popped from the stack and the value is written to the output port
p2s	A value is read from the input port and the value is pushed onto the stack
sleep	Put the processor into sleep mode

Table 3-1 Instructions in the subset

Our proposed design is a 16-bit asynchronous java processor that can directly execute some useful instructions defined in [10] with 16-bit operands including register read/write operations, arithmetic operations, logical operations and branch operations. The rest of them including method invocation, method return, array operation and object manipulation are handled by software routines. The summary of the subset is given in table 3-1.

Exception handling is not implemented in the java processor. Also, the data type of the operands for all instructions is fixed for 16-bit. So, the operation for integers (32-bit) can be simplified to the operation for short integers. (16-bit)

According to [10], the return value of the instruction Ret is read from a local variable. Instead, this return value is read from the stack in our design in order to simplify the architecture of the processor. Also, for ishl and ishr, 'n' should be the lower four bits of the value on the top of the stack. The instructions s2p and p2s that are not defined in [10] are added into the instruction set of the processor for manipulating the I/O port. Also, the instruction sleep is added to the instruction set to support the sleep mode.

### 3.2 Architecture of the java processor

- |  |
|--|
| <ul style="list-style-type: none"><li>● 16-bit ALU</li><li>● 16X16-bit RAM</li><li>● 10 level 16-bit Stack</li><li>● 16-bit address bus</li><li>● One 16-bit input port and one 16-bit output port</li></ul> |
|--|

Table 3-2 hardware features

In order to simplify the design, we have considered the processor as a 24-bit fixed-length instruction set processor executing instructions with an 8-bit opcode and a 16-bit operand. The hardware features are shown in table 3-2.

The java processor is a stack-based processor. So, its performance is limited by frequent stack access. The technique of instruction folding used in picoJava design of

Sun Microelectronics [11][12] can alleviate this problem. By using this technique, more than one instruction are decoded before execution. Then a single operation is used to represent the function of these instructions when necessary. However, this is too complicated to realize in the java processor core for the contactless smart card. Instead, we use result-forwarding technique in order to increase the performance of the processor. So, the data written to the stack is forwarded when the next instruction is reading the element from the top of stack and this saves one write-stack operation.

Figure 3-2 shows a simplified pipelined architecture of the asynchronous java processor. Only the primary data path from the fetch unit to the ALU is depicted. Other secondary data paths such as the link between the operand latch and the stack are omitted for clarity. Also, the latch controllers of all the latches and the control circuits of the request/acknowledge signal are not shown. Because it is a fixed length instruction set processor, a simpler instruction decoder can be used to decode different types of instruction. After reset, all latches and flip-flops have been initialized and the processor is ready to operate. Since we use the normally opaque latch controller, the latches of all functional blocks are closed before their latch controllers receive the request signals. They will open and then close again to store the input data after their latch controllers have received the request signals.

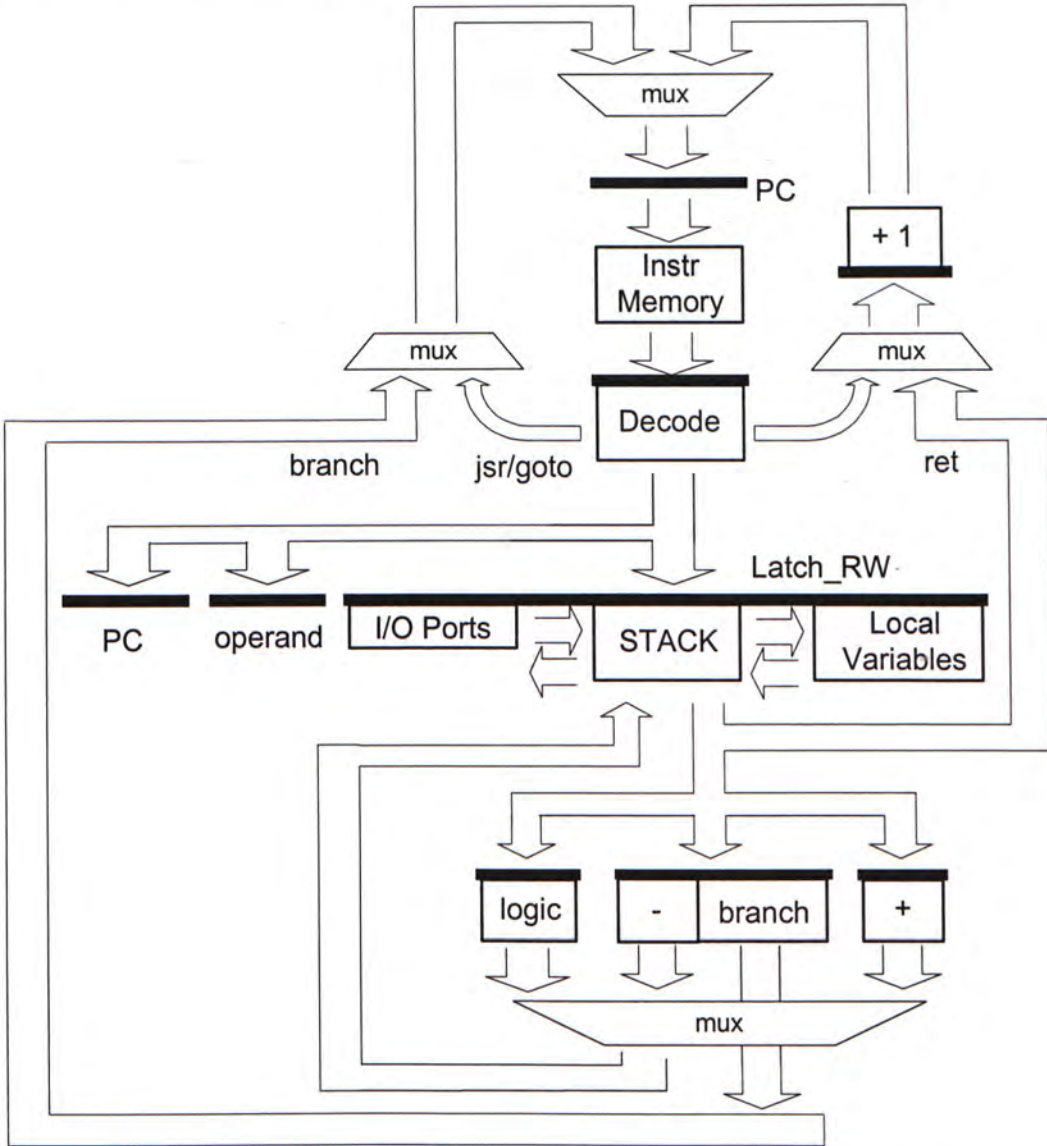


Figure 3-2 Block diagram of asynchronous java processor (black boxes represent latches)

When the latch for storing the PC (the top most latch) is updated, an instruction is fetched from the instruction memory. Then both the PC and instruction are stored in the input latch of the decoder. After the instruction is decoded, a request signal is passed to the latch controller (not shown) of the Latch\_RW and then the control bits for manipulating the read/write operation of the stack, block of local variables and the I/O ports are stored in the Latch\_RW. Also, other request signals are passed to other functional blocks optionally depending on the decoded instruction at the same time.

So, the fork III can be utilized to distribute the request signals. Since there are four other request signals that can be transmitted by the decoder, the fork element that can generate five request signals as shown in figure 3-3 is used in this situation.

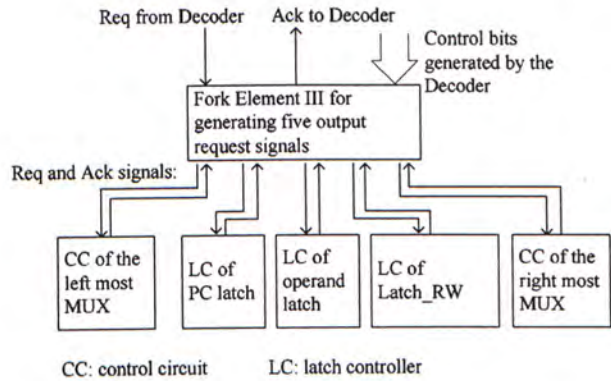


Figure 3-3 Fork of request signals generated by the Decoder

These four other requests are transmitting to (1) control circuit of the left-most multiplexer shown in figure 3-2 for updating the top-most latch that is used to store the program counter (The new value of the program counter is the result of addition of the old value of the program counter and the 16-bit operand fetched from the instruction memory) if the instruction is a goto or a jump subroutine operation, (2) control circuit of the right-most multiplexer for updating the top-most latch that is used to store the program counter if the current instruction is not a goto, unconditional branch, jump subroutine or return operation, (3) the latch controller of the PC latch (the left-most latch for storing the address of the decoded instruction), (4) the latch controller of the operand latch.

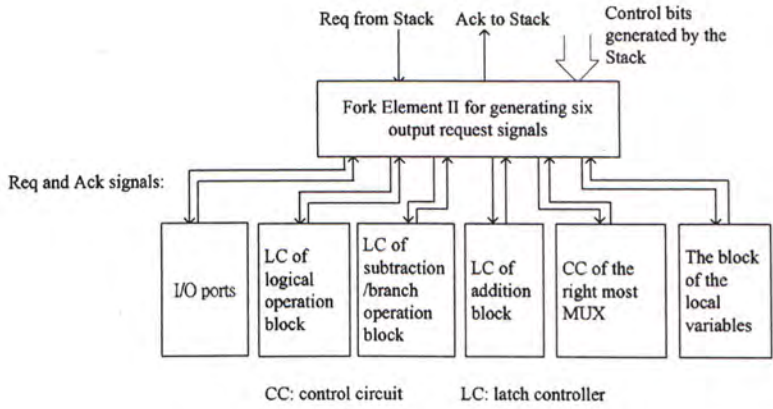


Figure 3-4 Fork of request signals generated by the Stack

The control circuit shown in figure 2-11 can realize the control circuits of the multiplexers shown in figure 3-2. The internal memory of the processor is represented by the Stack (10 level stack) and the block of local variables (16X16bit RAM). The stack can provide operands for the ALU (addition block, subtraction/branch block and logical operation block), input data of the block of local variables, updated data of the top most PC latch shown in figure 3-2 or updated data for the output ports. Since the output data of the stack will be passed only to one of them for implementing an instruction, the fork element II can realize this fork of request signals and the Req/Ack signals are shown in Figure 3-4. Also, the data from the PC latch/operand latch, the result of the ALU, the local variables and the data read from input port can be written to the stack. The variable index (address of the memory) of a local variable is provided by the operand latch. Data can be passed from the stack to the block of local variables or from this block to the stack directly without passing through any latches or functional blocks. The top most latch shown in figure 3-2 can also be updated by a new address of branch instruction (result of addition of operand and PC) through the left-most data path.

### 3.3 Basic building blocks of the java processor

#### 3.3.1 Block of local variables

The 16X16bit RAM of the java processor realizes the block of local variables. The address of the RAM (variable index) is provided by the operand latch shown in figure 3-2.

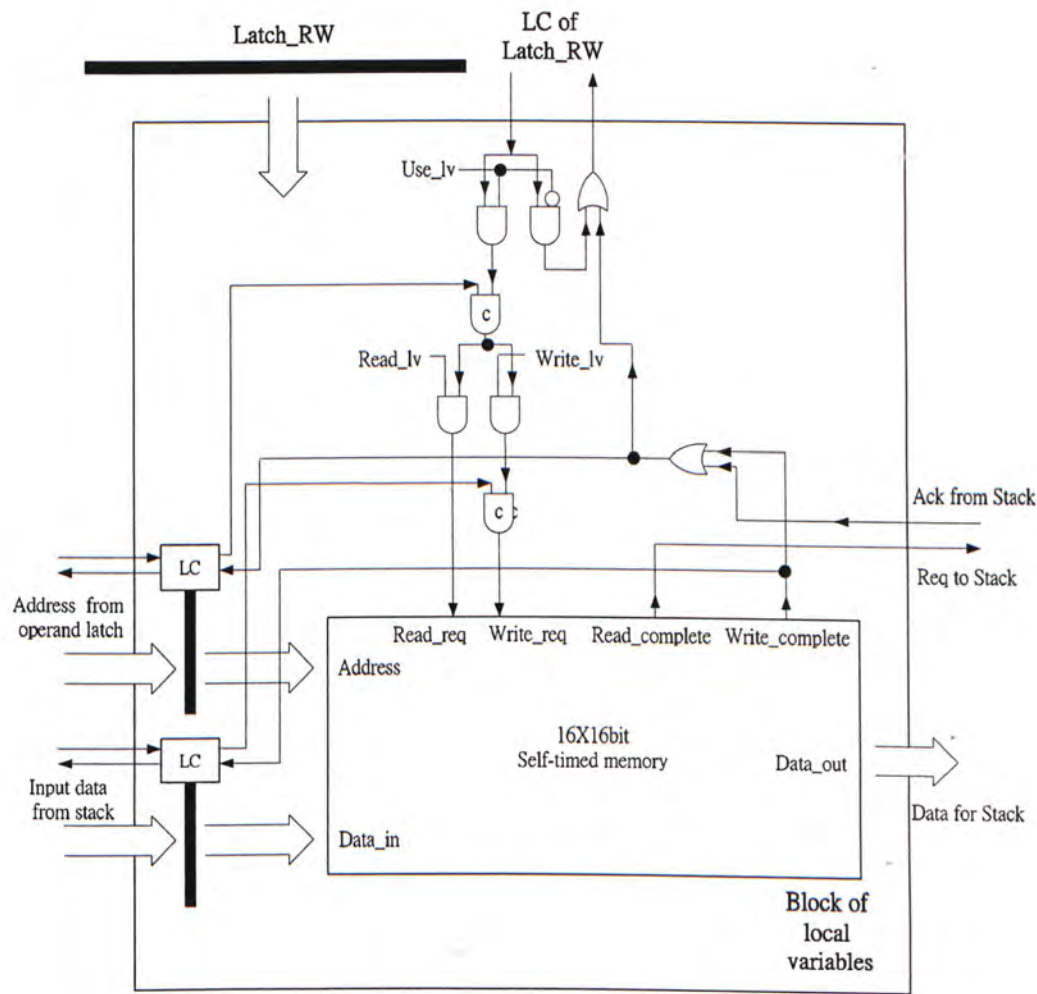


Figure 3-5 Schematic of block of local variables (black boxes represent latches)

The schematic of the block of local variables is shown in the figure 3-5. A 16X16bit self-timed memory is used. The structure of the block of local variables is very simple

because the input data written to the block of local variables can only be read from the stack and the output data read from the block of local variables can only be written to the stack. Also, the address of self-timed memory (the variable index of the local variable) can only be read from the operand latch shown in figure 3-2.

Two control bits, Read\_lv and Write\_lv, are read from the Latch\_RW. If the instruction involves a read operation from a local variable, the control bit Read\_lv is set to 1. Otherwise, it is 0. Similarly, if the instruction involves a write operation to a local variable, the control bit Write\_lv is set to 1. Otherwise, it is 0. In the processor design, both read operation and write operation appeared in a single instruction are not supported. So, Read\_lv and Write\_lv are never set to 1 at the same moment. The other control bit Use\_lv is used to indicate whether the current instruction involves a read operation from the block of local variables or a write operation to it. When one of the Read\_lv or Write\_lv is 1, Use\_lv is set to 1. Otherwise, it is 0. When Use\_lv is 0, the request signal to the block of local variables will reflect to the LC of Latch\_RW.

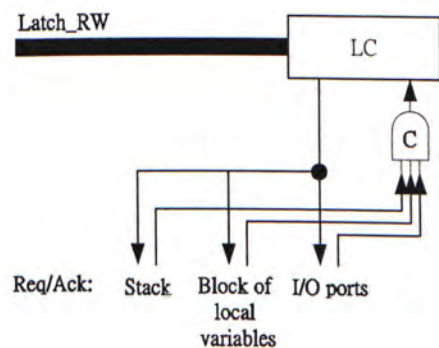


Figure 3-6 Request signals transmitted by the latch controller of the Latch\_RW

As mentioned before, the Latch\_RW shown in figure 3-5 provides control bits for the read or write operation of the block of local variables, stack and I/O ports. So, the fork

I for transmitting three request signals (and receive three acknowledge signals) as shown in Figure 3-6 is used.

3.3.2 Stack

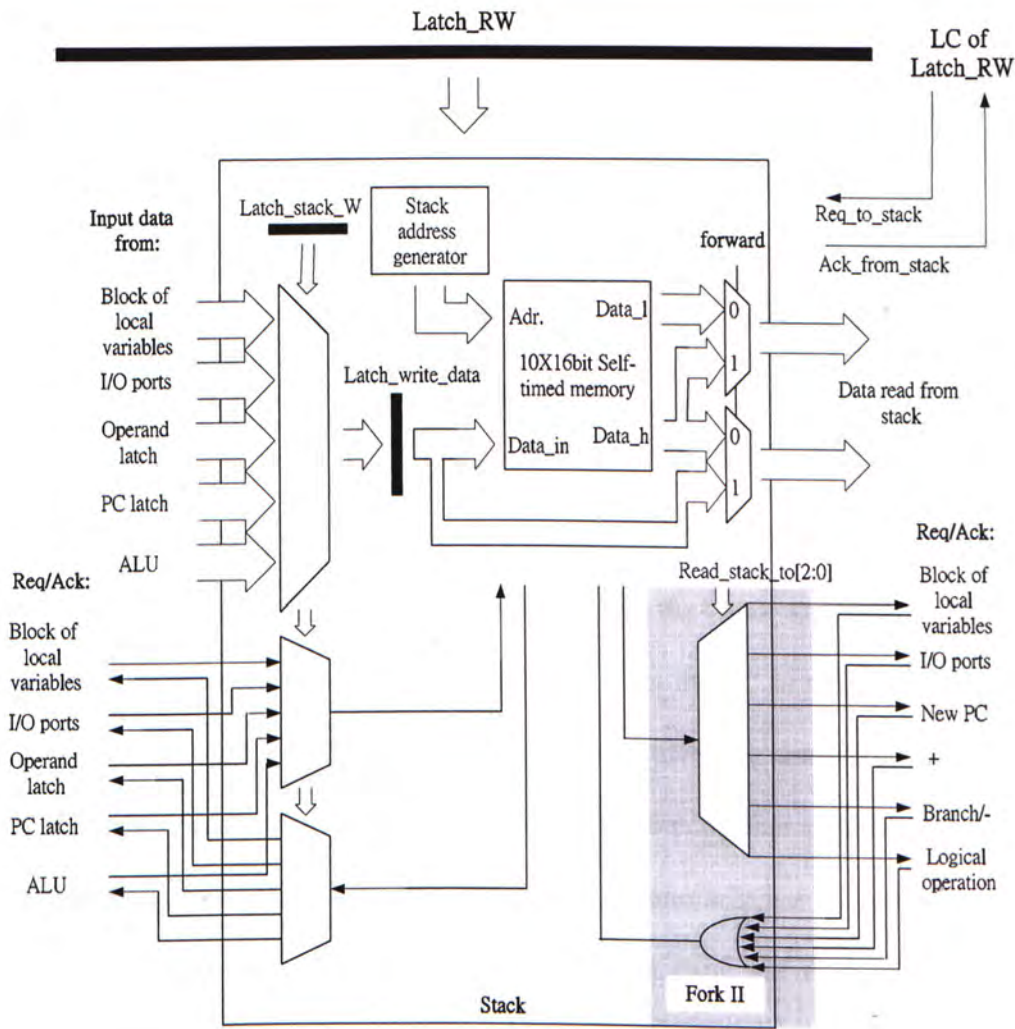


Figure 3-7 Stack (black boxes represent latches)

Stack is a very important component in the java processor since all the operands of the ALU operations are read from it and it can be a temporary register for data transfer. The simplified structure of the stack is shown in figure 3-7. Some control signal paths for the request and acknowledge signals are not shown and only some important

control bits are shown. It can support 3 types of operations: 1) a single read from the stack in an instruction, 2) a single write to the stack in an instruction and 3) both a read operation and a write operation in an instruction. The stack supports the write operation of one data, the read operation of one data and the read operation of two data. Also, the result-forwarding technique is used in the processor design.

If the request signal from the LC of Latch\_RW is received and the instruction involves a write operation to the stack (a control bit Write\_stack that is not shown is 1), some control bits read from the Latch\_RW are stored into the Latch\_stack\_W. The input data written to the stack can be read from one of five functional blocks and it is selected by the data stored in the Latch\_stack\_W.

When the request signal for one of these five functional blocks is transmitted to the stack, the data written to the stack will firstly store into the Latch\_write\_data shown in figure 3-7. After receiving the next request signal from the LC of the Latch\_RW, the next instruction is executed. If the next instruction involves the read operation of one data, the data stored in the Latch\_write\_data will be directly forwarded as the output data read from the top of the stack and the stack pointer will not be updated. If the next instruction involves the read operation of two data, the data stored in the Latch\_write\_data will be forwarded and a data is read from the self-timed memory. (And then the stack pointer is updated)

If the next instruction does not involve a read operation, the data stored in the Latch\_write\_data will be stored into the 10X16bit self-timed memory and the stack pointer (address of the self-timed memory) will be updated.

The read operation from the stack is quite similar to that from the block of local variables. However, the read data from the stack can be written to one of six functional blocks. Fork element II (as shown before in figure 3-4) shown in figure 3-7 can transmit a request signal to one of the six functional blocks controlled by control bits `Read_stack_to[2:0]` read from the `Latch_RW`. The read operation starts when the stack has received a request signal from the latch controller of the `Latch_RW` and a control bit `Read_stack` is 1 (not shown). Since result-forwarding technique is used in the processor design, the data can be read from the `Latch_write_data` or the self-timed memory. Then a new value of the stack pointer is generated by the stack address generator when the data (one or two) is read from the self-timed memory.

When the instruction involves both a read operation and a write operation, (both `Read_stack` and `Write_stack` are 1) the write operation does not start until the read operation from the stack is completed.

### **3.3.3 Self-timed RAM**

Two self-timed RAM is used in the java processor. One of them is for the stack (10X16bit) and the other one is for the block of local variables (16\*16bits). Delay element is used to detect the completion signal of the write operation of these self-timed memories. For the read operation, the completion can be easily detected by monitoring the bit lines of the column of the self-timed RAM

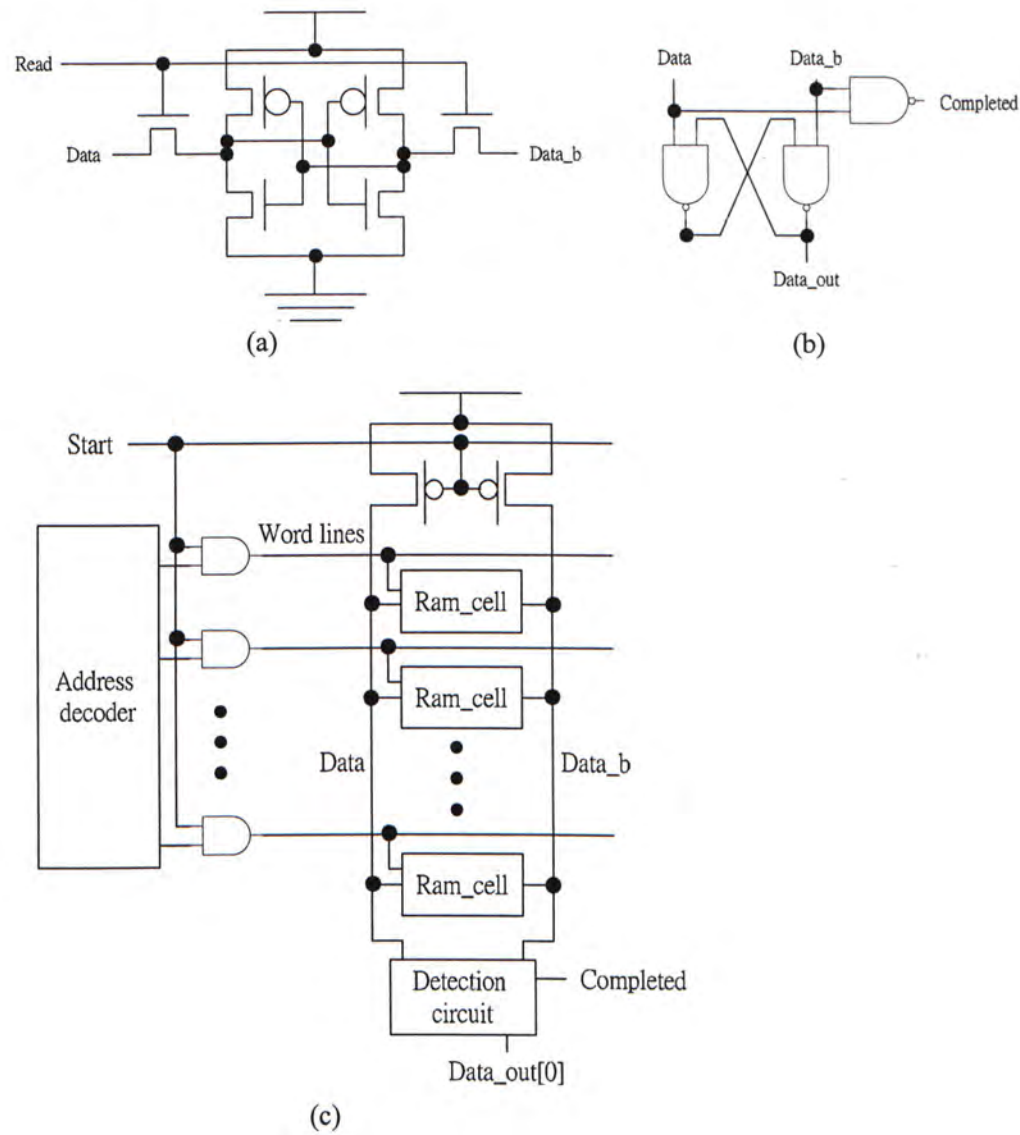


Figure 3-8 (a) Ram cell (b) Detection circuit (c) A column of the RAM

The Ram cell and the detection circuit are shown in figure 3-8(a) and 3-8(b) respectively. The detection circuit can both store the data read from the stack and detect the completion of the read operation. A column of the RAM is shown in figure 3.8(c). When the signal 'Start' is 0, the signals on the bit lines of the column (Data and Data\_b) are precharged to 1. Also, the signal 'Completed' of the detection circuit is 0. When the read operation starts, one of Data and Data\_b is changed from 1 to 0. Then the signal 'Completed' is changed to 1 and the output data 'Data\_out' can be read

from the stack. After Data\_out is read by other functional block, the ‘Start’ will change back to 0 and both signals Data and Data\_b are precharged to 1 again. Then the signal ‘Completed’ is changed back to 0. (The data read from the stack before is stored on ‘Data\_out’). Since the memory size of self-timed memories in the processor are very small, no sense amplifier is used.

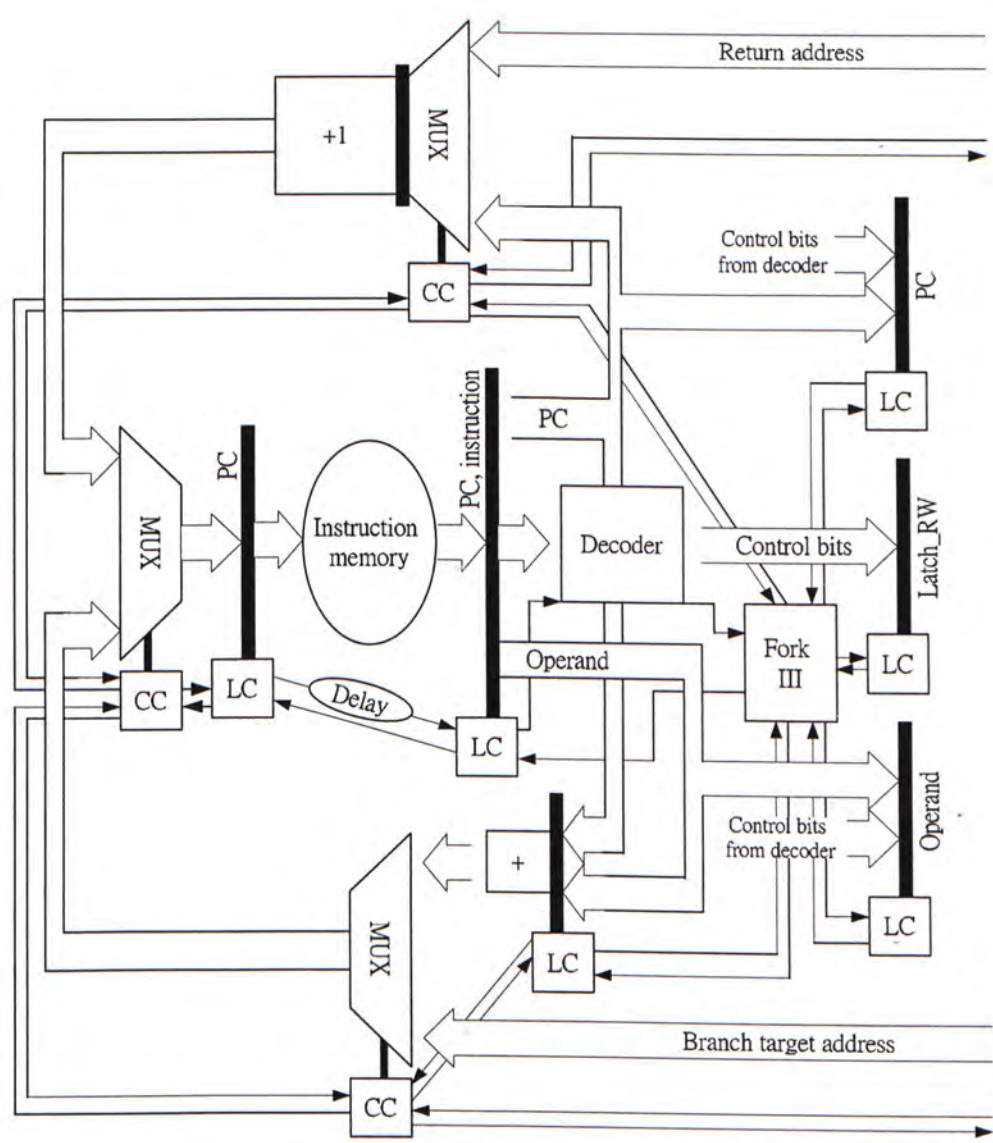


Figure 3-9 Fetch/decode unit (black boxes represent latches)

3.3.4 Fetch/decode unit

The complete schematic of the fetch/decode unit including all the data paths and request/acknowledge signals is shown in figure 3-9. LC is latch controllers and CC is the control circuit of the multiplexer as shown before in figure 2-11.

The data width of the program counter (PC) is 16 bits and that of the instruction is 24 bits including an 8-bit opcode and a 16-bit operand. The decoder is a lookup table for generating control bits. A delay element (not shown) in the decoder is used to generate the completion signal for the handshake protocol.

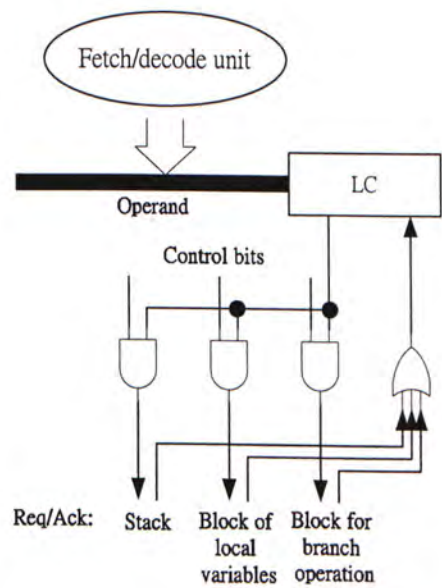


Figure 3-10 Request signals from the latch controller of the operand latch

The Latch\_RW is used to store the control bits for the stack, the block of local variables, the I/O ports and the ALU. Since the data stored in the PC latch and operand latch can pass to two or more functional blocks as shown in figure 3-10, some control bits generated by the decoder is also written to them.

3.3.5 Self-timed adder

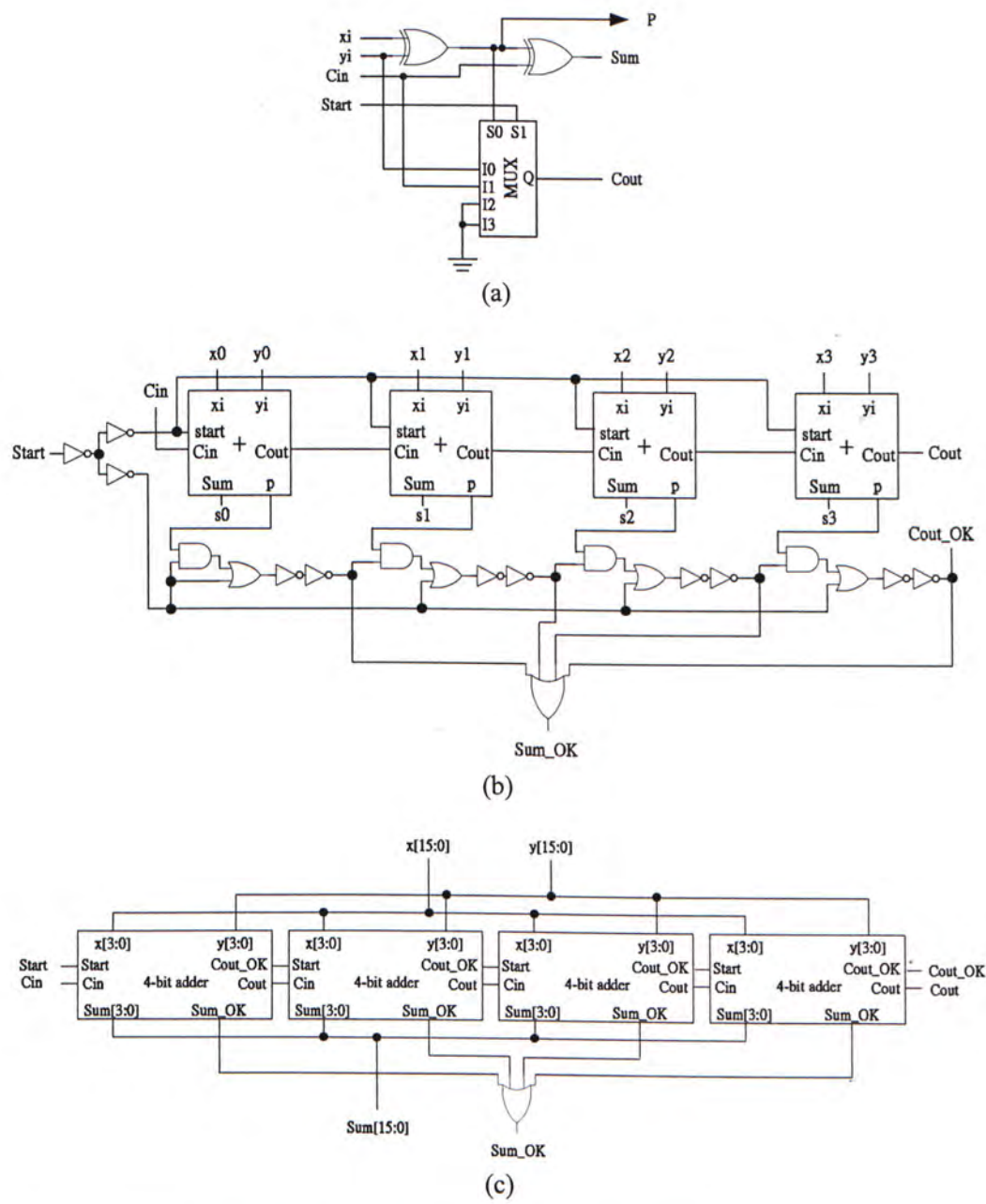


Figure 3-11 (a) Basic adder cell (b) 4-bit adder (c) 16-bit adder

The self-timed adders used in the java processor are based on [7]. The basic adder cell is shown in figure 3-11(a). The signal ‘Start’ is the start signal for the operation and

the signal 'P' is used to detect the completion for generating the carry out bit 'Cout'. After reset, the Start is 1 and the Cout is 0. Before the Start is changed from 1 to 0, the input bits  $x_i$ ,  $y_i$  and  $C_{in}$  are valid and the signal 'P' is generated. If P is 0, the values of  $x_i$  and  $y_i$  are both equal to 0 or both equal to 1. In this situation, the value of the carry out signal 'Cout' is equal to that of  $y_i$  (or  $x_i$ ) for any value of the signal  $C_{in}$ . So, the Cout can be calculated without waiting for the completion of the generation of  $C_{in}$ . However, if P is 1, Cout cannot be calculated without the valid value of  $C_{in}$ .

The 4-bit adder is shown in figure 3-11(b). When the signal Start changes from 1 to 0, the operation starts. Cout\_OK is the completion signal for generating Cout and Sum\_OK is the completion signal for generating the sum. The 16-bit adder shown in figure 3-11(c) is realized by four 4-bit adders

### 3.4 Token flow

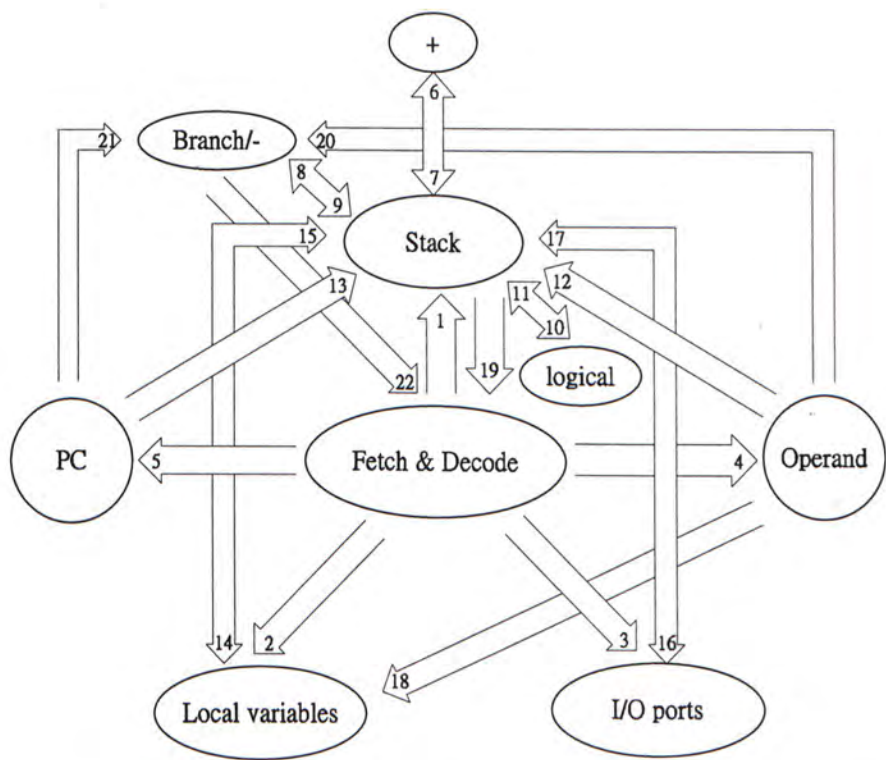


Figure 3-12 Paths for transmitting control bits and data bits (path 1, path 2 and path 3 are the paths for transmitting control bits)

The paths for transmitting control bits and data bits in the java processor are shown in figure 3-12. Since the asynchronous circuit design technique is used, all the functional blocks shown in this figure operate only when necessary. As mentioned before, Req and Ack (not shown) are bundled with these paths to control the communication between the functional blocks.

Instruction	Paths involved
sipush const	1, 4, 12
iload index	1, 2, 4, 18, 15
istore index	1, 2, 4, 18, 14
iadd	1, 6, 7
isub	1, 8, 9

ishl	1, 10, 11
ishr	1, 10, 11
iand	1, 10, 11
ior	1, 10, 11
ixor	1, 10, 11
if_scmpeq_w offset	1, 4, 5, 8, 20, 21, 22
if_scmplt_w offset	1, 4, 5, 8, 20, 21, 22
if_scmpge_w offset	1, 4, 5, 8, 20, 21, 22
goto offset	None
jsr offset.	1, 5, 13
ret	1, 19
nop	None
s2p	1, 3, 16
p2s	1, 3, 17
sleep	None

Table 3-3 Paths for each instruction

The paths for all instructions are shown in table 3-3. There is no data transfer when executing goto, nop and sleep since these instructions are only executed in the fetch-decode unit. In order to show how they work, some examples are given as follows:

- iload index:

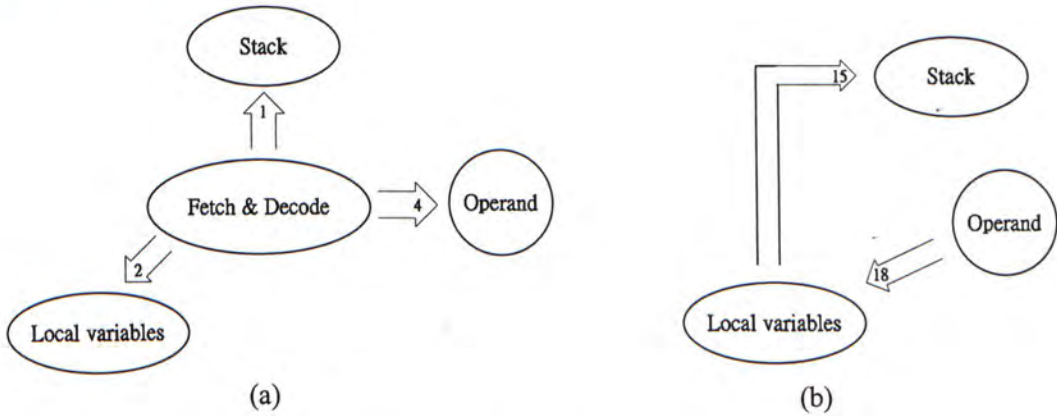


Figure 3-13 Execution of 'iload index': (a) Step 1 (b) Step 2

Step1: Control bits are transmitted to the stack, block of local variables and the operand latch. The variable index of the local variable is also transmitted to

the operand latch. (Figure 3-13(a))

Step2: The data stored in operand latch (variable index) is transmitted to the block of local variables. Then the data of the selected local variable is push onto the stack. (Figure 3-13(b))

● sipush const:

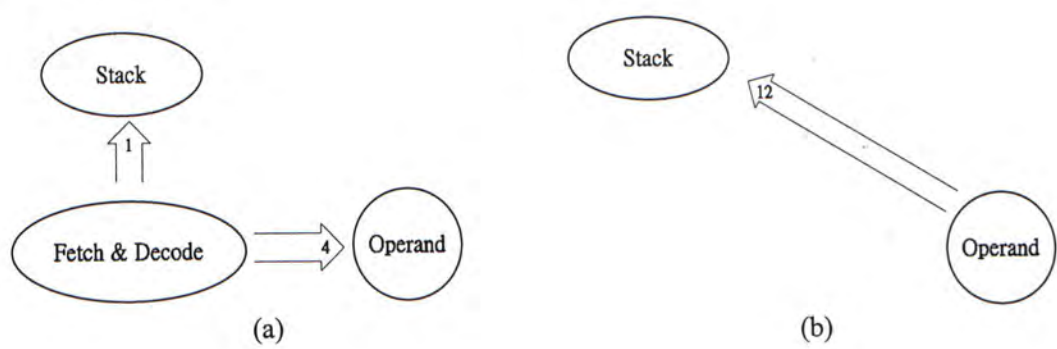


Figure 3-14 Execution of 'sipush const': (a) Step 1 (b) Step 2

Step1: Control bits are transmitted to the stack and the operand latch. An immediate data for the stack is also transmitted to the operand latch. (Figure 3-14(a))

Step2: The data stored in the operand latch is push onto the stack. (Figure 3-14(b))

● iadd:

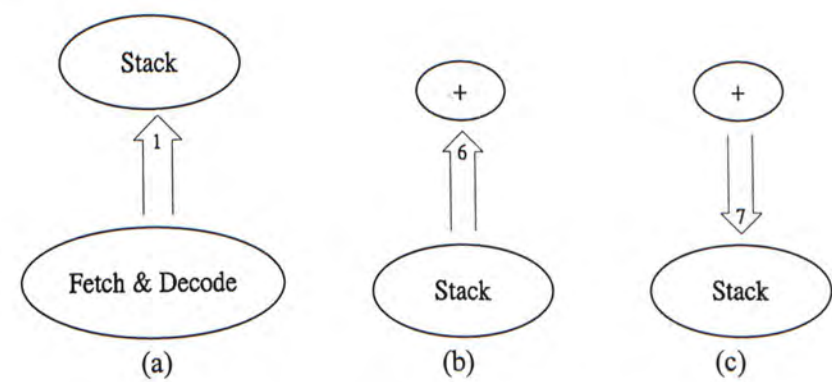


Figure 3-15 Execution of 'iadd': (a) Step 1 (b) Step 2 (c) Step 3

- Step1: Control bits are transmitted to the stack. (Figure 3-15(a))
- Step2: Two data are popped from the stack and transmitted to the block of addition. (Figure 3-15(b))
- Step3: The result of addition is pushed onto the stack. (Figure 3-15(c))

● if\_scmpeq\_w offset:

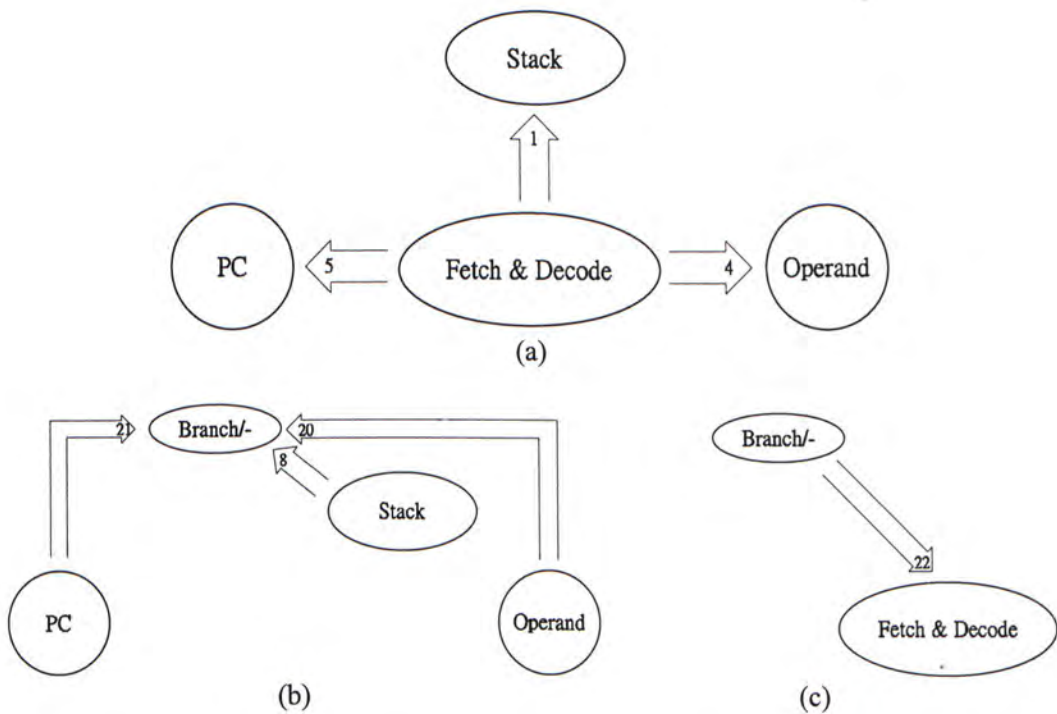


Figure 3-16 Execution of 'if\_scmpeq\_w offset': (a) Step 1 (b) step2 (c) Step3

- Step1: Control bits are transmitted to the stack, the operand latch and the PC latch. Also, offset address for the branch operation is transmitted to the operand latch and the PC of the current instruction is transmitted to the PC latch. (Figure 3-16(a))
- Step2: The data stored in the operand latch and the data stored in the PC latch is

transmitted to the block of branch operation. Two data popped from the stack is also written to this block and the branch target address is calculated. (Figure 3-16(b))

Step3: The branch target address is written to the latch for storing the new PC in the fetch/decode unit. (Figure 3-16(c))

Chapter 4

Results and Discussion

4.1 Simulation Results of test programs

Three test programs for executing all the instructions are used to evaluate the performance of the java processor.

*Program 1:*

Address	Instruction	
00	nop	
01	p2s	Input port is set to 0003
02	istore	00
03	iload	00
04	sipush	0001
05	isub	
06	istore	00
07	iload	00
08	s2p	
09	iload	00
0A	sipush	0000
0B	if_scmpeq_w	0002
0C	goto	FFF7
0D	sipush	FFFF

0E                    s2p  
0F                    sleep

Result:  
Data in the output port: 0002, 0001, 0000, FFFF

Program 2:

Address	Instruction:	
00	p2s	Input port is set to 0006
01	sipush    0001	
02	ishl	
03	sipush    0002	
04	ishr	
05	sipush    000F	
06	ixor	
07	sipush    00C8	
08	ior	
09	sipush    006F	
0A	iand	
0B	s2p	
0C	sleep	

Result:  
Data in the output port: 004C

Program 3:

Address	Instruction:	
00	jsr	0004
01	sipush	0001
02	s2p	
03	sleep	
04	p2s	Input port is set to 0002
05	sipush	0003
06	if_scmlt_w	0003
07	sipush	0002
08	s2p	
09	sipush	0003
0A	sipush	0004
0B	if_scmpge_w	0004
0C	sipush	0003
0D	s2p	
0E	ret	
0F	sipush	0004
10	s2p	
11	ret	

Result:

Data in the output port: 0003, 0001

Program	MIPS	Power (mW)	MIPS/W
1	25	11.12	2248
2	28	12.667	2210
3	23	10.412	2209

Table 4-1 Simulation Results

The simulation results of the three programs done by software spectreS are shown in table 4-1. However, the simulation is not accurate since all parasitic resistors and capacitors of the metals connecting the logic cells are not included in this simulation. Obviously, the actual performance can be much worse than the simulation result shown in Table 4-1. So, measuring the fabricated chip is necessary for giving a better evaluation of the performance.

The java processor is organized by the standard library cells and the memory cells for the self-timed memories. The processor design is placed and routed by Silicon Ensemble. Also, a post layout Verilog-XL simulation is done for verification after replacing all my library cells with standard library cells. For example, the memory cells are replaced with flip-flops. This simulation result shows that the processor works at 9MIPS when executing the bytecodes in program 1.

4.2 Experimental result

Supply(V)	MIPS	Power (mW)	MIPS/W
3.3	18.7	14.1	1326
3	17.2	10.8	1593
2.5	14.2	6.3	2254
2	10.2	2.5	4080

Table 4-2 Experimental Results

The java processor chipset for testing is fabricated by 0.35um CMOS process. The bytecodes of program 1 are stored in the rom of the java processor. The experimental result shown in Table 4-2 is done by executing the bytecodes in the test program.

For a contactless smart card system, the external clock is 13.56MHz [13] and the power consumption has to be less than 30mW. [14] The experimental result shows that the power consumption of the processor is 14.1mW when the speed is 18.7 MIPS. (At 3.3V) Obviously, this can fulfill the requirement of the system of contactless smart card. We can reduce the power consumption by decreasing the supply voltage. Also, its power consumption is about equal to zero in the power down mode.

Supply voltage = 2.5V				
	CMOS technology	MIPS	Power (mW)	MIPS/W
Asynchronous QDI 8-bit microcontroller [13]	0.25um	23.8	28.0	850.3
Our design: 16-bit asynchronous java processor	0.35um	14.2	6.3	2254

Table 4-3 Result comparison

The result comparison of two processor designs is shown in table 4-3. The processor for comparison is an asynchronous quasi-delay insensitive (QDI) 8-bit microprocessor [13] fabricated with 0.25  $\mu\text{m}$  technology. Both the architecture and the instruction set of the two processors are different. However, they are also designed for the system of contactless smart card.

When comparing the MIPS/W figures, the performance of our proposed design is about 2.7 times better than that of the asynchronous QDI 8-bit microcontroller.

### **4.3 Future work**

#### **4.3.1 Architecture and instruction set**

In order to increase the efficiency of the java processor, the instructions pop, pop2, iinc, neg, iushr, if\_scmlp\_w, if\_scmplt\_w, saload and astore are suggested to be added into the subset. Among them, saload and astore are two very important instructions for accessing the memory and they are necessary for the java processor.

Also, as mentioned before, the definition of the instruction ret in the java processor is different from that in [10]. So, we can modify the architecture of the java processor so that the function of this instruction is the same as that defined in [10]

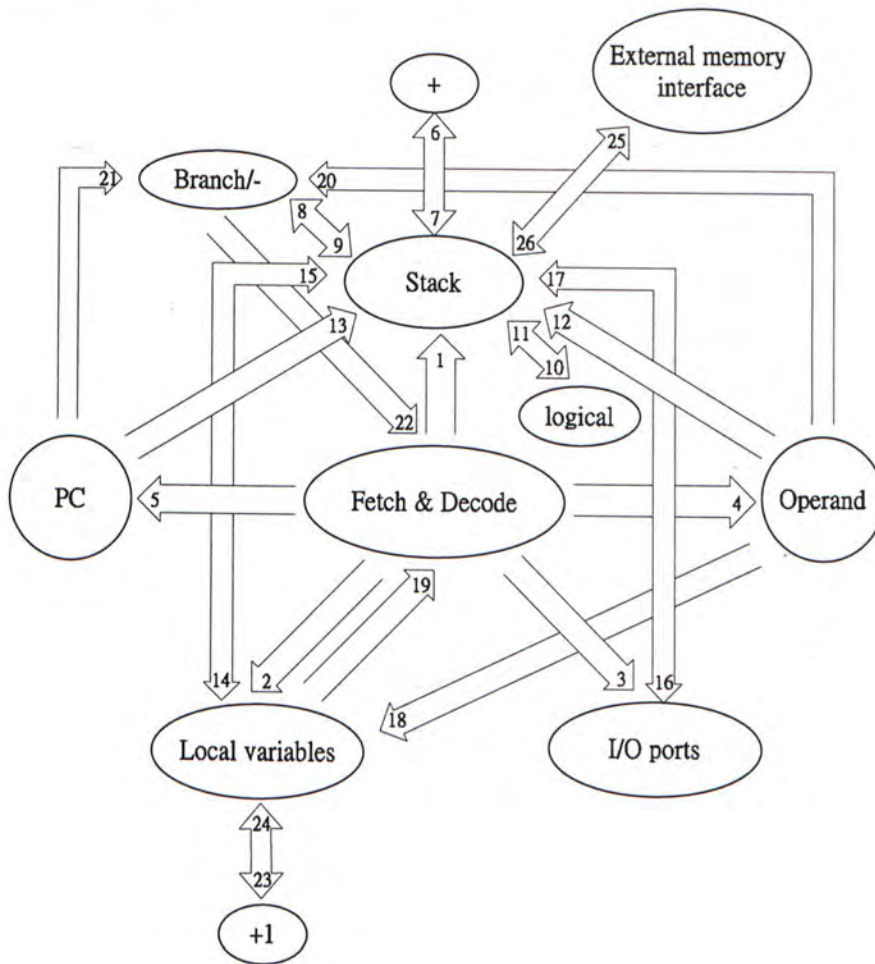


Figure 4-1 modified architecture of the java processor

The modified architecture of the java processor is shown in figure 4-1. The data path 23 and the data path 24 are used for the instruction iinc. Also, in order to support the instructions saload and sastore, an external memory interface is added into the design.

### 4.3.2 Interrupts

Interrupt is currently not implemented in the java processor. External interrupt and timer interrupt are essential for communication in the smart card system. These interrupts can also be handled in the fetch/decode unit.

### **4.3.3 Java program**

A complete java program for the smart card system is needed to be developed in order to evaluate the performance of the java processor. The java program should be able to communicate with the card reader, access the memory and perform encryption/decryption of the input and the output data. Since some instructions defined in [10] cannot be executed directly in the java processor, some software subroutines have to be written for executing the unsupported instructions.

## Chapter 5

### Conclusion

The architecture of a low power asynchronous java processor for contactless smart card is designed and it can directly execute the java bytecodes in a subset of the instruction set defined in the Java Card Virtual Machine specification. They are including the register read/write operations, ALU operations and branch operations. The remaining java bytecodes are handled by software routines.

The java processor core are organized by the fetch unit, decode unit, stack, the block of local variables and the ALU. It is a stack-based processor and the result-forwarding technique is used in the stack in order to save some write operations to the stack. Also, self-timed adders and self-timed memories are used in order to increase the operating performance of the asynchronous java processor.

In this design, we use four-phase bundled-data protocol for normally opaque latch controller as the handshaking protocol in the asynchronous pipeline. The control circuits of the asynchronous pipelines used in the java processor including fork, join and merge are introduced. Also, a new delay element is proposed in order to reduce the power consumption of the asynchronous pipeline.

The java processor chipset is fabricated using 0.35um CMOS process. The power

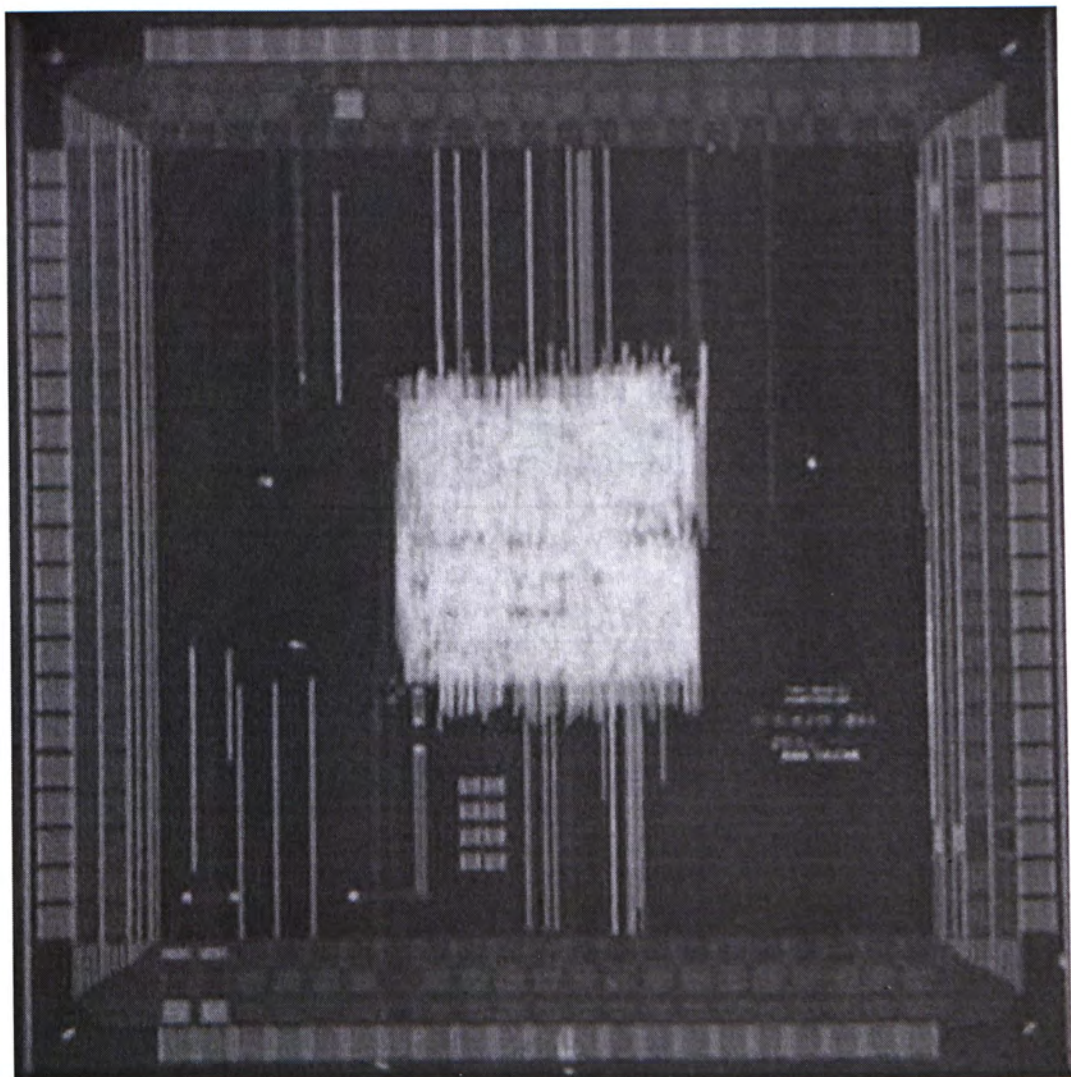
consumption of the chipset is 14.1mW when the operating speed is 18.7 MIPS and it is suitable for the application of contactless smart card. Also, its power consumption is about equal to zero in the power down mode.

Finally, in order to increasing the operation performance of the java processor, some suggestions of the hardware and software design are given.

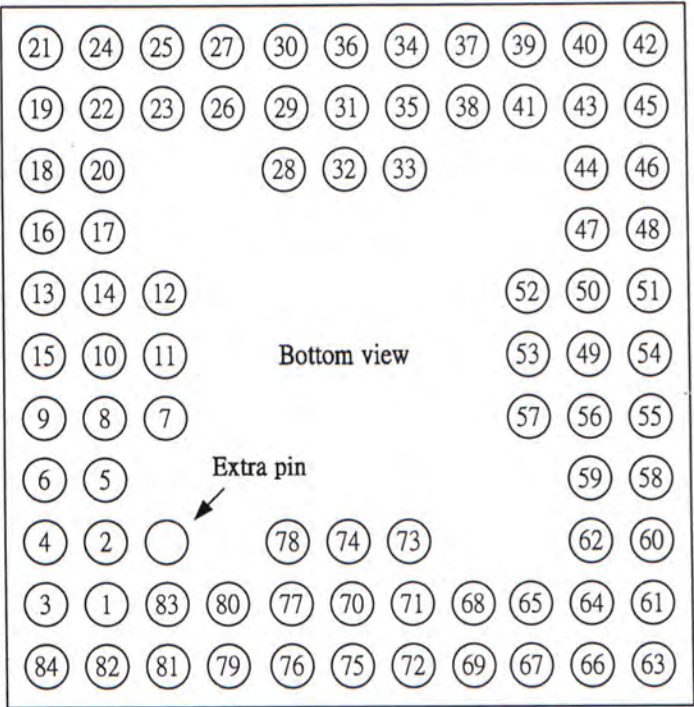
# Appendix

## Chip micrograph for the java processor core

(Size 1.2 mm<sup>2</sup>)



Pin assignment of the java processor



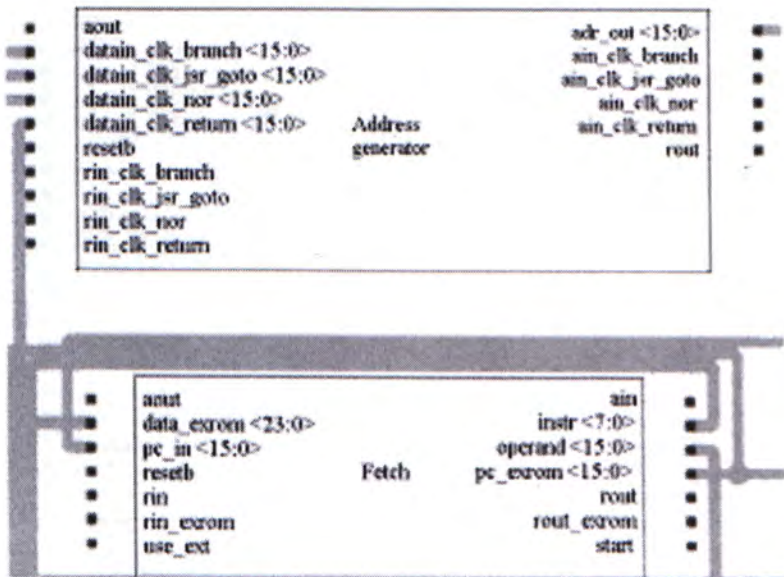
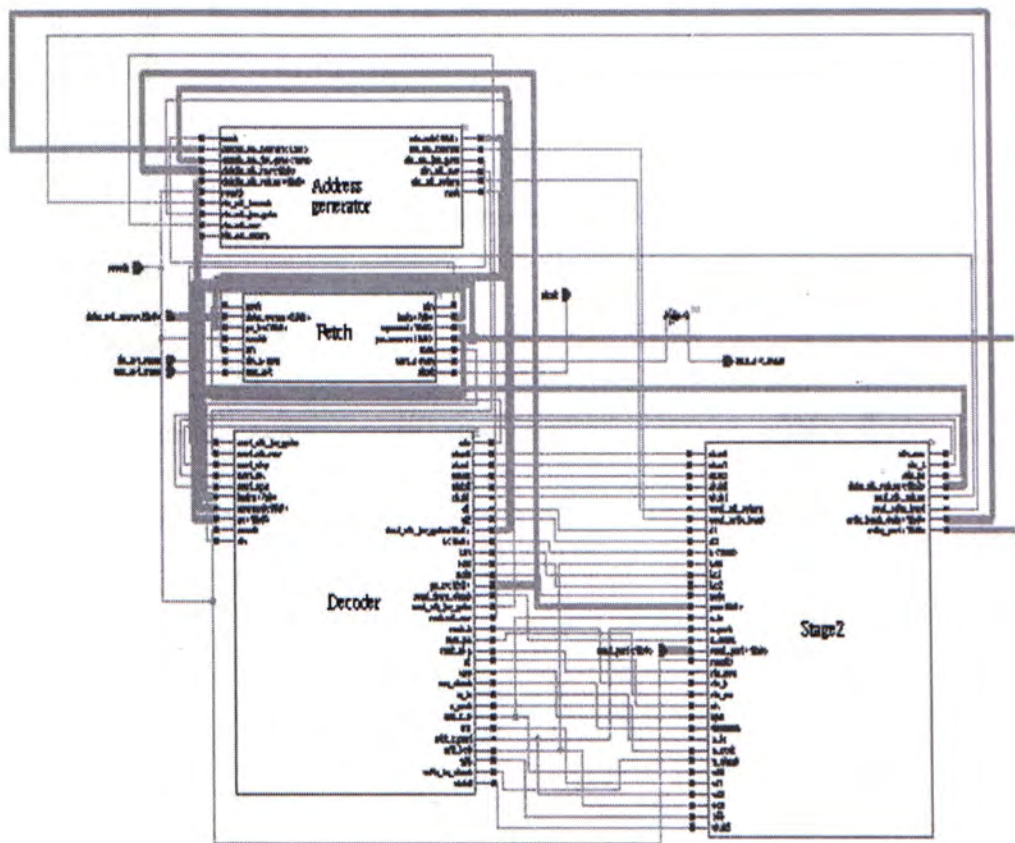
Pin number	Pin Name	IN/OUT	Description
1	GND	IN	
2	GND	IN	
3	Start	IN	Start signal
4	Use_ext_mem	IN	Enable the external instruction memory
5	Rin_ext_mem	IN	Request signal from the external instruction memory
6	Rout_ext_mem	OUT	Request signal to the external instruction memory
7	VDD	IN	
8	Rport [0]	IN	Bit 0 of the read port
9	Rport [1]	IN	Bit 1 of the read port
10	Rport [2]	IN	Bit 2 of the read port
11	Rport [3]	IN	Bit 3 of the read port
12	Rport [4]	IN	Bit 4 of the read port
13	Rport [5]	IN	Bit 5 of the read port
14	Rport [6]	IN	Bit 6 of the read port
15	Rport [7]	IN	Bit 7 of the read port
16	Rport [8]	IN	Bit 8 of the read port

17	Rport [9]	IN	Bit 9 of the read port
18	Rport [10]	IN	Bit 10 of the read port
19	Rport [11]	IN	Bit 11 of the read port
20	Rport [12]	IN	Bit 12 of the read port
21	Rport [13]	IN	Bit 13 of the read port
22	Rport [14]	IN	Bit 14 of the read port
23	Rport [15]	IN	Bit 15 of the read port
24	VDD	IN	
25	Wport [0]	OUT	Bit 0 of the write port
26	Wport [1]	OUT	Bit 1 of the write port
27	Wport [2]	OUT	Bit 2 of the write port
28	Wport [3]	OUT	Bit 3 of the write port
29	Wport [4]	OUT	Bit 4 of the write port
30	Wport [5]	OUT	Bit 5 of the write port
31	Wport [6]	OUT	Bit 6 of the write port
32	Wport [7]	OUT	Bit 7 of the write port
33	Wport [8]	OUT	Bit 8 of the write port
34	Wport [9]	OUT	Bit 9 of the write port
35	Wport [10]	OUT	Bit 10 of the write port
36	Wport [11]	OUT	Bit 11 of the write port
37	Wport [12]	OUT	Bit 12 of the write port
38	Wport [13]	OUT	Bit 13 of the write port
39	Wport [14]	OUT	Bit 14 of the write port
40	Wport [15]	OUT	Bit 15 of the write port
41	GND	IN	
42	Adr_ext_mem [0]	OUT	Bit 0 of the address bus of the external instruction memory
43	Adr_ext_mem [1]	OUT	Bit 1 of the address bus of the external instruction memory
44	Adr_ext_mem [2]	OUT	Bit 2 of the address bus of the external instruction memory
45	Adr_ext_mem [3]	OUT	Bit 3 of the address bus of the external instruction memory
46	Adr_ext_mem [4]	OUT	Bit 4 of the address bus of the external instruction memory
47	Adr_ext_mem [5]	OUT	Bit 5 of the address bus of the external instruction memory
48	Adr_ext_mem [6]	OUT	Bit 6 of the address bus of the external instruction memory
49	Adr_ext_mem [7]	OUT	Bit 7 of the address bus of the external instruction memory
50	Adr_ext_mem [8]	OUT	Bit 8 of the address bus of the external instruction memory
51	Adr_ext_mem [9]	OUT	Bit 9 of the address bus of the external instruction memory
52	Adr_ext_mem [10]	OUT	Bit 10 of the address bus of the external instruction memory
53	Adr_ext_mem [11]	OUT	Bit 11 of the address bus of the external instruction memory

54	Adr_ext_mem [12]	OUT	Bit 12 of the address bus of the external instruction memory
55	Adr_ext_mem [13]	OUT	Bit 13 of the address bus of the external instruction memory
56	Adr_ext_mem [14]	OUT	Bit 14 of the address bus of the external instruction memory
57	Adr_ext_mem [15]	OUT	Bit 15 of the address bus of the external instruction memory
58	VDD	IN	
59	VDD	IN	
60	Data_ext_mem [0]	IN	Bit 0 of the data bus of the external instruction memory
61	Data_ext_mem [1]	IN	Bit 1 of the data bus of the external instruction memory
62	Data_ext_mem [2]	IN	Bit 2 of the data bus of the external instruction memory
63	Data_ext_mem [3]	IN	Bit 3 of the data bus of the external instruction memory
64	Data_ext_mem [4]	IN	Bit 4 of the data bus of the external instruction memory
65	Data_ext_mem [5]	IN	Bit 5 of the data bus of the external instruction memory
66	Data_ext_mem [6]	IN	Bit 6 of the data bus of the external instruction memory
67	Data_ext_mem [7]	IN	Bit 7 of the data bus of the external instruction memory
68	Data_ext_mem [8]	IN	Bit 8 of the data bus of the external instruction memory
69	Data_ext_mem [9]	IN	Bit 9 of the data bus of the external instruction memory
70	Data_ext_mem [10]	IN	Bit 10 of the data bus of the external instruction memory
71	Data_ext_mem [11]	IN	Bit 11 of the data bus of the external instruction memory
72	Data_ext_mem [12]	IN	Bit 12 of the data bus of the external instruction memory
73	Data_ext_mem [13]	IN	Bit 13 of the data bus of the external instruction memory
74	Data_ext_mem [14]	IN	Bit 14 of the data bus of the external instruction memory
75	Data_ext_mem [15]	IN	Bit 15 of the data bus of the external instruction memory
76	Data_ext_mem [16]	IN	Bit 16 of the data bus of the external instruction memory
77	Data_ext_mem [17]	IN	Bit 17 of the data bus of the external instruction memory
78	Data_ext_mem [18]	IN	Bit 18 of the data bus of the external instruction memory
79	Data_ext_mem [19]	IN	Bit 19 of the data bus of the

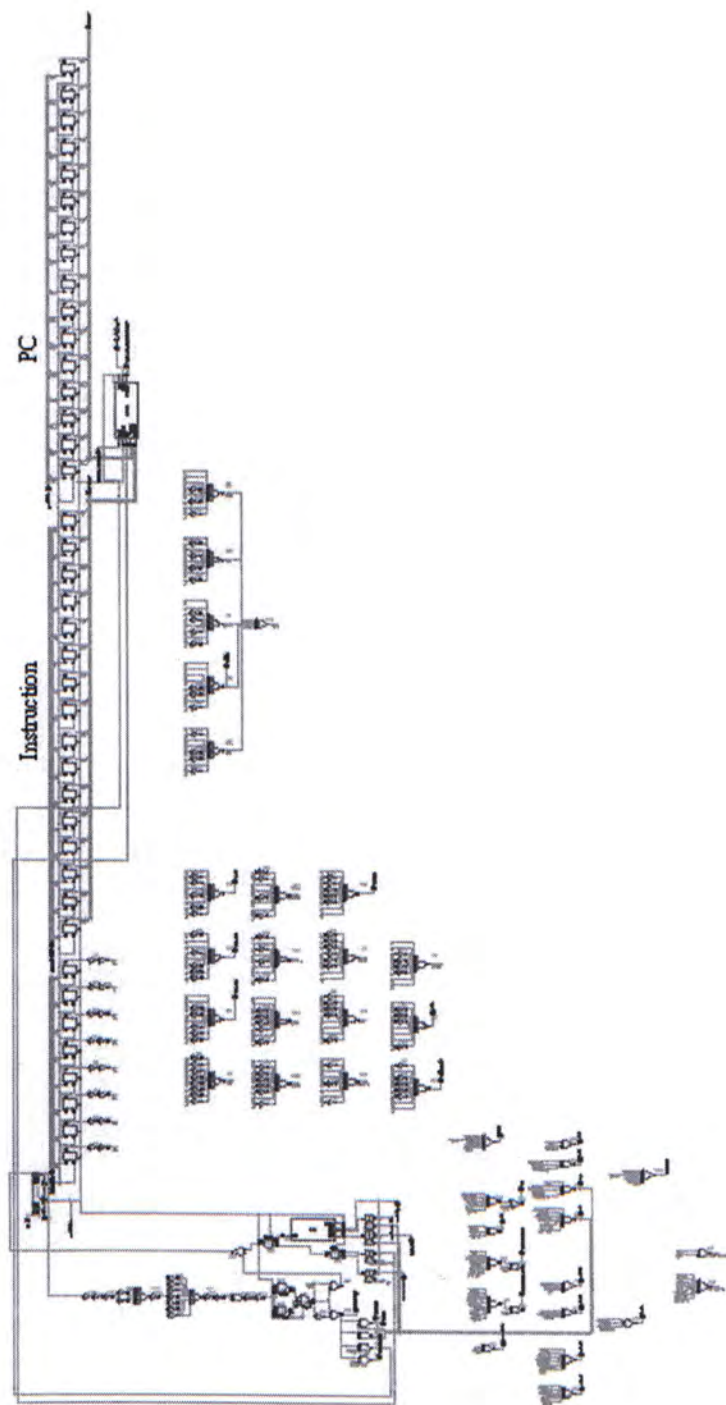
			external instruction memory
80	Data_ext_mem [20]	IN	Bit 20 of the data bus of the external instruction memory
81	Data_ext_mem [21]	IN	Bit 21 of the data bus of the external instruction memory
82	Data_ext_mem [22]	IN	Bit 22 of the data bus of the external instruction memory
83	Data_ext_mem [23]	IN	Bit 23 of the data bus of the external instruction memory
84	Resetb	IN	Reset signal

Schematic of the java processor

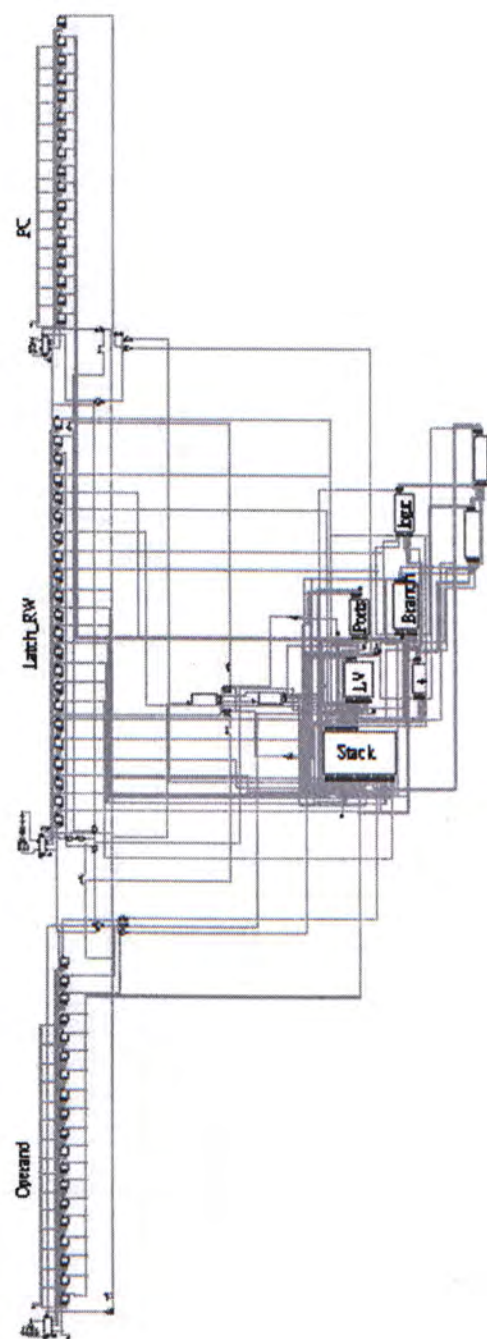




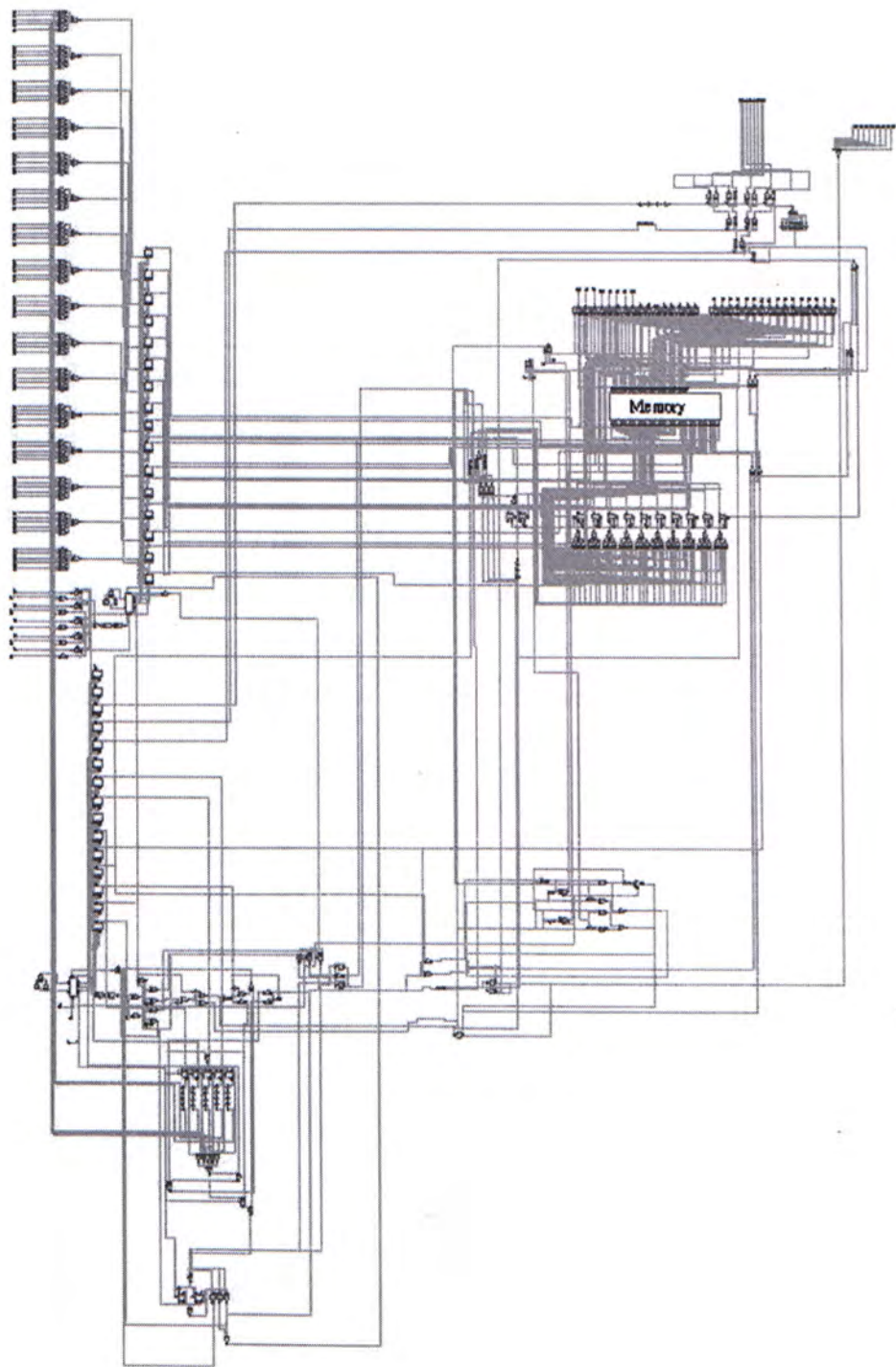
Schematic of the decoder



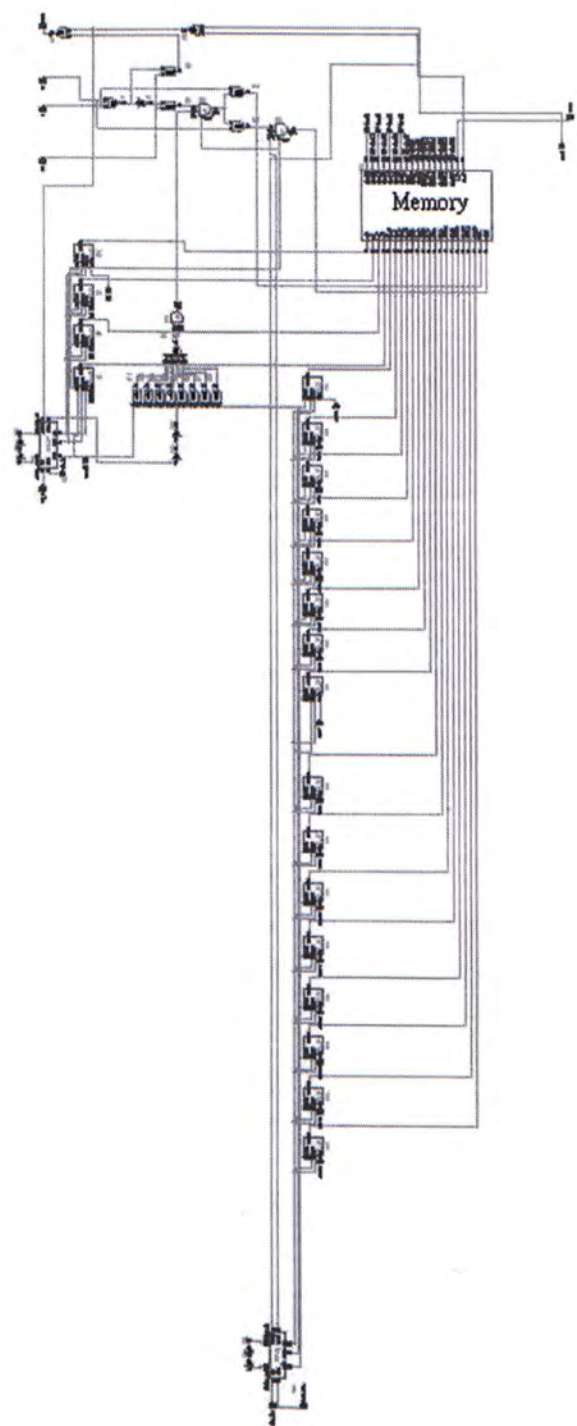
Schematic of the Stage2 of the java processor



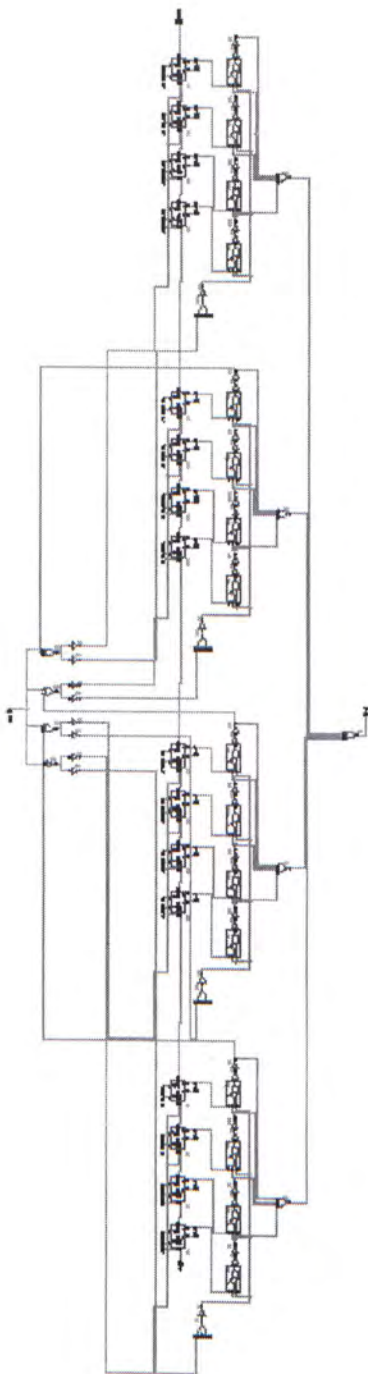
Schematic of the stack



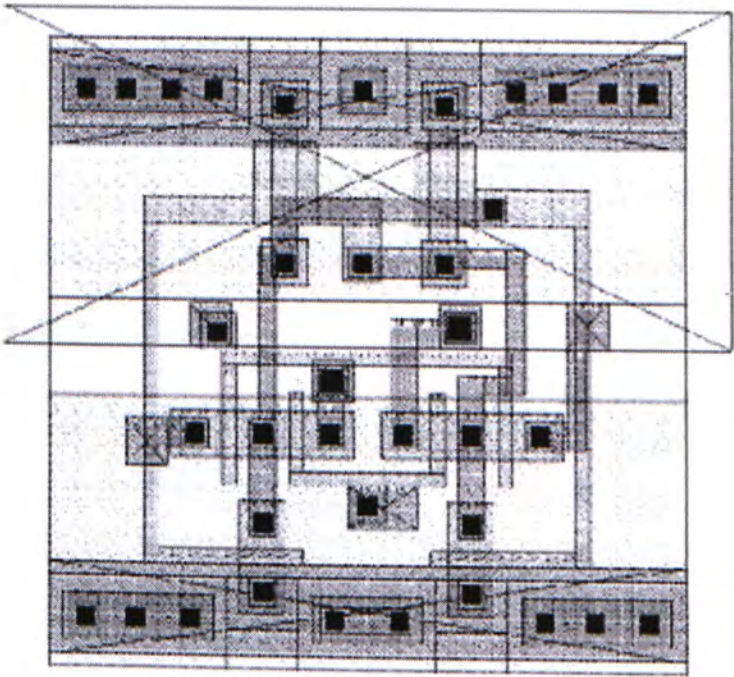
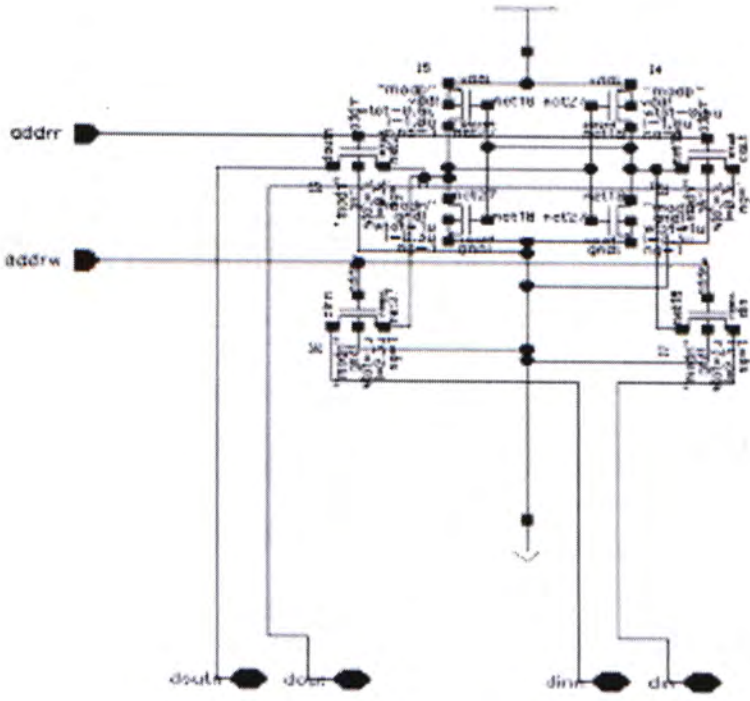
Schematic of the block of the local variables



Schematic of the 16-bit self-timed adder



The schematic and the layout of the memory cell



## Reference

- [1] Endecott, P.B, "Superscalar instruction issue in an asynchronous microprocessor", Computers and Digital Techniques, IEE Proceedings, Volume: 143 Issue: 5, Sept.1996, Pages 266 –272
- [2] Jens Sparsø, Steve Furber, "Principles of Asynchronous Circuit Design", Kluwer Academic Publishers, 2001
- [3] Java Card Special Interest Group, "2 Smart Card Overview", [http://www.javacard.org/others/smart\\_card.htm](http://www.javacard.org/others/smart_card.htm).
- [4] Zhang Jianjie, Li Feihui, Ge Yuanqing, Yue Zhenwu, Yang Zhilian, "A Java processor suitable for applications of smart card", ASIC, 2001. Proceedings. 4th International Conference, 2001, Pages: 736-739
- [5] Taylor, G.S., Blair, G.M., "Reduced complexity two-phase micropipeline latch controller", Solid-State Circuits, IEEE Journal of , Volume: 33 Issue: 10 , Oct 1998, Page(s): 1590 -1593
- [6] I. E. Sutherland, "Micropipelines", Commun. ACM, vol. 32, pp. 720-738, June 1989.
- [7] Perri, S., Corsonello, P., Cocorullo, G., Cappuccino, G., Staino, G., "VLSI implementation of a fully static CMOS 56-bit self-timed adder using overlapped execution circuits", Electronics, Circuits and Systems, 2001. ICECS 2001. The 8th IEEE International Conference on , Volume: 2 , 2001, Page(s): 723 -727 vol.2
- [8] Woods, J.V., Day, P., Furber, S.B., Garside, J.D., Paver, N.C., Temple, S., "AMULET1: an asynchronous ARM microprocessor", Computers, IEEE Transactions on , Volume: 46 Issue: 4 , Apr 1997, Page(s): 385 -398

- [9] K. T. Christensen, P. Jensen, P. Korger, and J. Sparsø, "The design of an asynchronous TinyRISC TR4101 microprocessor core", In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, Pages 108-119. IEEE Computer Society Press, 1998
- [10] Sun Microsystems, "Java Card 2.2 Virtual Machine Specification", <http://java.sun.com/products/javacard>, 2002
- [11] M. O'Connor, H. McGhan, "PicoJava: A Direct Execution Engine for Java Bytecode", IEEE Computer, 31, 10, 22, 1998
- [12] Lee-Ren Ton, Lung-Chung Chang, Min-Fu Kao, Han-Min Tseng, Shi-Sheng Shang, Ruey-Liang Ma, Dze-Chaung Wang, Chung-Ping Chung, "Instruction folding in Java processor", Parallel and Distributed Systems, 1997. Proceedings, Pages 138-143
- [13] Abrial, A., Bouvier, J., Renaudin, M., Senn, P., Vivet, P., "A new contactless smart card IC using an on-chip antenna and an asynchronous microcontroller", Solid-State Circuits, IEEE Journal of , Volume: 36 Issue: 7 , Jul 2001, Page(s): 1101 -1107
- [14] Cormie C., Grimonprez G., "A RISC microprocessor for contactless smart cards", EUROMICRO 97. 'New Frontiers of Information Technology', Proceedings of the 23rd EUROMICRO Conference , 1-4 Sep 1997 Page(s): 658 -663



CUHK Libraries



004076698