JAVA-BASED MICROPROCESSOR

by

Mohammad Faiz bin Md. Khuzaimah
2836

Dissertation submitted in partial fulfilment of
the requirements for the
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

JUNE 2006

Universiti Teknologi PETRONAS
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

JAVA-BASED MICROPROCESSOR

by

Mohammad Faiz bin Md. Khuzaimah

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfilment of the requirement for the
BACHELOR OF ENGINEERING (Hons)
(ELECTRICAL & ELECTRONICS ENGINEERING)

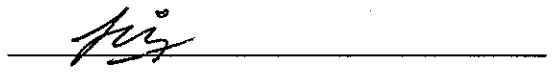
Approved by,

(PATRICK SEBASTIAN)

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK
June 2006

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgments, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.



(MOHAMMAD FAIZ BIN MD. KHUZAIMAH)
Student ID: 2836

ABSTRACT

Java-based Microprocessor is a project aimed to develop a processor that implements Java virtual machine (JVM) instruction set into the hardware. The objective of the project is enabling a Java application to be executed without the need of JVM, but in a more specific term, it is aimed to be an alternative non commercial processor as a supporting base for educational research and development of embedded systems. With the current application of Java, the Java Runtime Edition (JRE), an inter medium Java OS, must be installed in every machine that is intended to execute Java *bytecode*. This proved to be inefficient, especially in embedded system where the resources are limited and upgrading is highly expensive.

The project was developed to be an easily comprehensible HDL, allowing others to pursue with advancement without complications. Thus, the HDL design were coded with behavioural style. In order to be more transparent for others to view the project development, the entire design is being developed by bottom-up approach. Four modules comprises the entire design – ALU, stacks, program counter and datapath. These modules were designed individually, allowing a separate test bench and test parameters, which also provided a better perspective of the microprocessor design.

The project has already progressed from an 8-bit processor in mind towards a 32-bit computer. The JVM has strict rules, allowing only certain instructions to execute with proper operands with the right data type. The project was not planned to allow operations of floating point number and doubles.

In conclusion, as for the use for supporting educational research and development, Java-based Microprocessor shall provide a solid foundation to embedded systems, where more enhancements would be needed before it can be utilized reliably.

ACKNOWLEDGMENT

First and foremost, all praises to Allah The Almighty that by His blessings I have been able to complete my final year project, the Java-Based Microprocessor. I would like to thank the following people who helped me in my final project.

Mr. Patrick Sebastian, my supervisor and Computer System Architecture lecturer, who came with the idea of this project and helped me with references projects and moral support all the way.

Mr. Lo Hai Hiung, a lecturer, who had gave me a good insight of HDL and Altera Quartus II.

My Parents, Mr. Md. Khuzaimah and Mrs. Hasnah, who has been very supporting, caring for my well-being and prayed for my success.

Mr. Faizan, a tutor, who had, taught me a good deal of HDL coding technique and introduction to Altera Quartus II

Dr. Yap Vooi Voon, a lecturer, for his critique of my project development.

I would also like to thank Nadirah Khairul Anuar, for her loving support every hard moments I endeavor while finishing my project.

TABLE OF CONTENTS

LIST OF ILLUSTRATION & TABLES

NOMENCLATURE

ALU	Arithmetic and Logic Unit
ASM	Algorithmic State Machine
CAD	Computer-aided Design
HDL	Hardware Description Language
JRE	Java Runtime Environment
JVM	Java Virtual Machine
LIFO	Last in First out
OS	Operating System
VHSIC	Very High Speed Integrated Circuit
MJava	Java-based Microprocessor (project title)

1. INTRODUCTION

1.1 BACKGROUND

Java applications have stormed the mobile industry lately, with current smart phones and mobile phones equipped with Java-enabled games and such. While the embedded systems industry is moving towards Java, there are several technical issues that prevent Java from being widely implemented in embedded devices such as set-top boxes, automotive systems and smart controllers.

The issues that prevent Java from being widely implemented are its performance and runtime execution efficiency. In order to execute a Java *bytecode*, the JRE must be running on top of a machine original operating system (OS) and this concept uses high resource. This has led to several developments of Java-based processor that is capable to execute the *bytecodes* without the need of JRE. These developments had been around since 1997 and one of the most Java processor was *picoJava*, designed by Sun Microsystems .

Java processor had been widely, and at the same time narrowly, developed to support embedded systems industry. Even in term education and research, there are many projects running that requires non commercial processor to support their development.

1.2 PROBLEM STATEMENT

The current concept of executing Java *bytecode* requires JRE to run on top of a machine OS. While using high resources, this also results in slow program load and unpredictable time-cycle execution. This drawback is considered trivial on personal computer, but in embedded systems and small devices such as handheld, the effect can be unacceptable.

Many Java processors being developed and many of them differ in

features and targeted media. Most of them were developed to suit medium-end to high-end small devices. In this project, the development focuses on the very basic of *bytecode* implementation and targeting only for embedded system with very limited resources.

Although the processor being developed in this project is a basic 32-bit signed integer, it is important to note that, in embedded systems application, building a complex and powerful processor is very costly. As a result, the processor in this project is devised to support fundamental features, dropping out the complex features that were entailed for higher performance systems and ensure that it will cater to embedded systems expeditiously.

1.3 OBJECTIVES & SCOPE OF STUDY

In general, Java-based Microprocessor (MJava) is aimed to implement JVM instruction set into a hardware stand-alone processor. In more specific term, it is aimed to be an alternative non commercial processor as a supporting base for the educational research and development of embedded systems.

In order to achieve the objective, some parameters had been refined and redefined, in which two of them are; to implement the JVM instruction with minimal use of external memory space; and keeping the final outcome as simple as possible with only the most basic requirement to execute Java class file properly and correctly.

2. LITERATURE REVIEW

2.1 PREVIOUS WORK ON JAVA PROCESSOR

Work on Java processor is not a new concept. It has been around since picoJava was initiated in 1998, but it is increasing in popularity. Several previous work had been used as references to the project. Each provided a different perspective on how to approach the solution.

2.1.1 Sun Microsystems' picoJava

picoJava is the first attempt on Java processor, developed under Sun Microsystems as the next step to popularize Java. Its advancement ideally suited the consumer electronic manufacturers need of small size processor core and high performance. It has been licenced to at least four (4) major companies.[1]

Its success in commercial values lies mostly on its high performance design computer architecture. The variable-sized cache, choice of with or without floating-point unit and the "stack register file" significantly improved performance. Its ability to execute legacy C/C++ as efficient as comparable RIS CPU is also a big advantage. **Figure 2.1** shows the architecture of picoJava cores, while the stack register file operation, treating file as a circular buffer is shown in **Figure 2.2**.

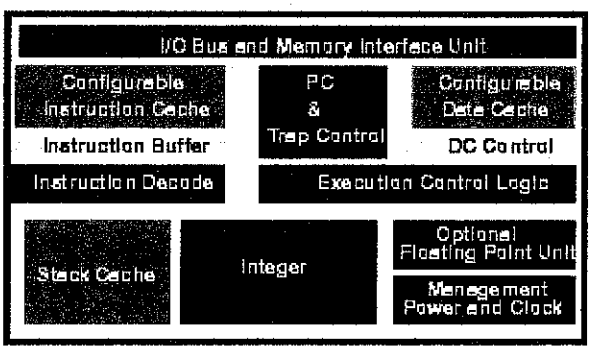


Figure 2.1: Block diagram of the picoJava cores[1]

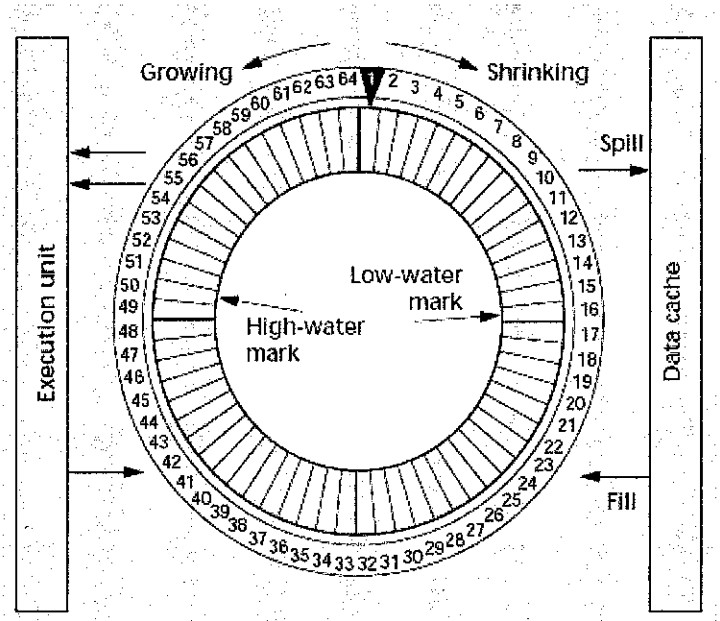


Figure 2.2: The picoJava core employ the circular register file to support stack-based processing[1]

2.1.2 Java Optimized Processor

Java Optimized Processor (JOP) was developed as part of a thesis project, focused on designing a processor for time-predictable execution of real-time tasks. Its primary implementation is in a field programmable gate array and the research demonstrates hardware implementation of the Java virtual machine results in a small design for resource-constrained devices. It had been designed to implement only the most frequently used instructions in the hardware level, while leaving the remaining to be executed on the software level.

In all measurements, JOP stated that the load of local variables and constants onto the stack accounts for more than 40% of instructions executed. This shows that an efficient realization of the local variable memory area, the stack and the transfer between these memory areas is mandatory. **On the other hand, the implementation of these three subjects, especially the stack, is critical to the project and thus, required.**[2]

JOP's own Java *bytecode* is named *microcode*. It is the native language for JOP. The microcode is translated from Java native language, *bytecodes* during execution, and both instruction sets are designed for an extended stack machine. In addition, JOP is fully pipelined architecture but with single cycle execution of microcode. It, however, used a fresh approach to mapping the Java *bytecode* to these instructions. **Figure 2.3** shows the data path of JOP, where it can be observed that the stack architecture allows for a short pipeline. This resulted in short branch delays.

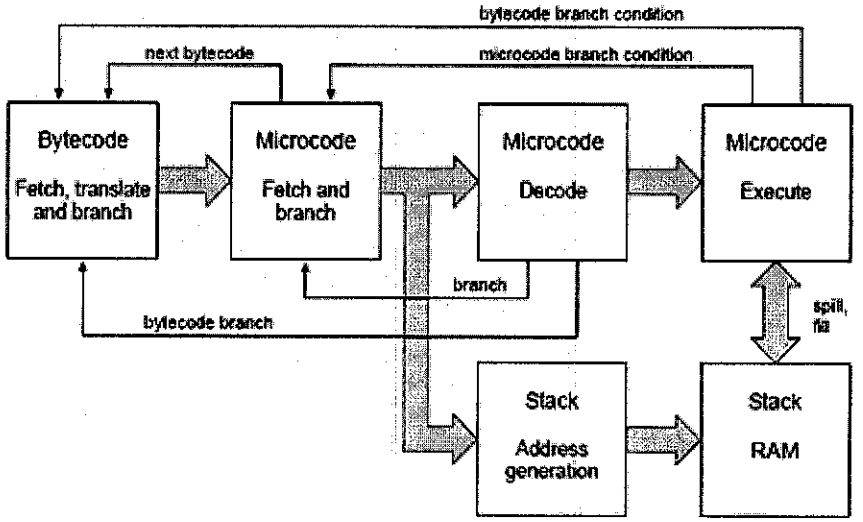


Figure 2.3: Block diagram of JOP cores[2]

2.1.3 Bernd Paysan's b16 Forth

The b16 processor is being developed as a Forth processor in an FPGA by Bernd Paysan. In this most brief summary, it has shown most promise as a better base to the project title Java-Based Microprocessor (MJava) that the JOP. Not only it is basically a stack-based processor, its minimalist design fits into small FPGA is most suitable for embedded systems application.

This processor is inspired by c18 from Chuck Moore, a popular forth processor, and is design entirely using Verilog HDL – a most convincing advantage for MJava side. Its basic processor architecture proved to be very

simplistic and practical for small application. Its stack machine was a radical approach but still has rooms for improvement.

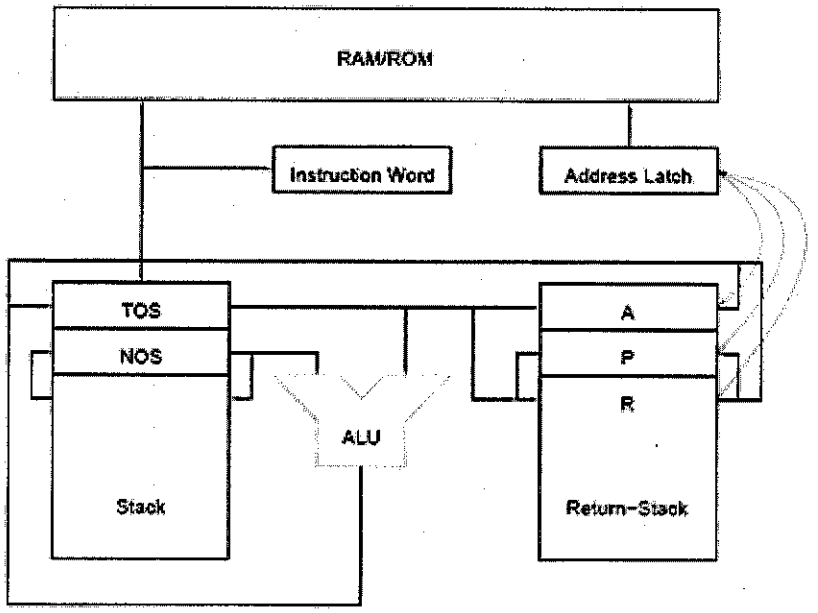


Figure 2.4: Block diagram of b16 cores[3]

2.2 JAVA VIRTUAL MACHINE

Java Virtual Machine (JVM) is an abstract computing machine, acting like a real computing machine, but executing Java *bytecode* instead of an assembler. It has an instruction set and capable of manipulating various memory areas at run time. JVM is also a stack-based machine in general, consisting several stacks for operands and return addresses. The stack-based JVM is further explained in subsection **2.3.1 JVM as Stack-based Machine.**

Java class file is translated into Java *bytecode*, which is used by JVM to be translated again into the specific native machine language. In short, JVM is a second layer operating system (OS) to the work station native OS, used in order to execute Java *bytecode*. The operation of *bytecode* basics is further explained in subsection **2.2.1 Fundamentals of Bytecode.**

JVM instructions consists of an opcode, which specify the operation to be performed, and followed by zero or more operands. This allow us to assume that implementing a complete JVM instruction set will result in exponentially increasing complexity, depending on the extent of how many instructions are being implemented. Certain JVM instructions can embody up to 14 operands each.

The JVM supports seven (7) primitives data types, listed in **Table 1**. Currently JVM consists of 202 instructions, although, many of the instructions are for similar operation but different data types involved. This was intended to make the *bytecodes* compact, by forcing opcodes to identify the data types involved instead of leaving it to the operands itself like in many other machine languages. (refer **Appendix D1** for a list of JVM opcodes with their corresponding hex values and **Appendix D2** for JVM opcodes with their relevant operand(s) type).[4][5]

Table 2.1: JVM primitive data types

Data Type	Definition
byte	one-byte signed two's complement integer
short	two-byte signed two's complement integer
int	4-byte signed two's complement integer
long	8-byte signed two's complement integer
float	4-byte IEEE 754 single-precision float
double	8-byte IEEE 754 double-precision float
char	2-byte unsigned Unicode character

2.2.1 Fundamentals of Bytecode

Bytecode is the machine language of the JVM. Since it was designed to be compact, bytcodes are fetched in streams. When an opcode reached the JVM, it indicates whether to encode zero or more operands

from the streams that immediately follow. Opcodes and operands in the *bytecodes* stream are aligned on byte boundaries, which means each opcode or operand is one byte of size. Operands of data type larger than a byte are broken into several bytes, stored in big-endian order in the *bytecodes* stream.

2.3 STACK MACHINE

Two major types of computer stack are Last-in First-out (LIFO) and First-in First-out (FIFO). While the latter act like a buffer, the former is being used vastly in main computing as a significant temporary storage, mainly to improve performance and to favour in compact machine code. LIFO stack by definition is conceptually the simplest way of storing information temporarily for use in common computation such as mathematical expression evaluation and recursive subroutine calling.

LIFO stacks can be constructed in software easily by allocating an array in memory and a variable with the array index number to keep track of the array position, known as stack pointer. The significant properties of LIFO stacks is the *push* and *pop* operations. A *push* will store information in the top most location (as defined by the stack pointer), while a *pop* extract information from the top most location to central processing (which later is deleted from the stack).

Stack-based machine or computer is increasingly becoming a favoured choice. Mostly due to its excellent mechanism of handling operations within procedures or recursive invocations. A nested branch and goto functions can be implemented very well with the use of LIFO stack. This also eliminates the need to specify location of return addresses, which could be space consuming.

2.3.1 JVM as Stack-based Machine

Computation in JVM centres on the stack to perform many operations, especially in arithmetics and returning from subroutines. In

JVM there are two separate stacks – operands stack and return stack. The latter was used strictly for return addresses, while the former is used for other information or operands. As Java *bytecode* was designed to be compact, many of the instructions are of zero operand. These instructions take values from the stacks. The stack will pop (read and delete) as many operands from the stacks as indicated by the opcode. The resultants are also usually *pushed* (stored) back onto the stacks.[4][5]

Assisting the stacks are the local variables, similar to working registers in many register-based machine. However, local variables use are limited to certain instructions and a programmer can barely manipulate this temporary storage. A number of instructions are dedicated for handling information between local variables and operands stack, but the direct use of local variables in calculation is unclear.

3. PROJECT WORK

A revised methodology presents several key changes in the project flow. Due to unforeseen delay caused by new findings, which led to new obstacles, and switch to Xilinx ISE, the hardware implementation on an FPGA kit has been deemed optional. In all, this project may end up as simulation-only if any of Xilinx FPGA is unavailable at the project disposal.

3.1 RESEARCH AND DESIGN APPROACH

Selecting and researching on Java ISA is not a direct precedence to project design and development. Still, it may provide key points to the direction of the development in term of the key elements that are necessary to be implemented.

Java ISA consists of 230 instructions, with three (3) reserved opcodes and 25 _quick opcodes. Nevertheless, current Sun JVM support only the 202 instructions (without the reserved and _quick opcodes) and many Java program had been written with these assumption. Thus, it is irrelevant to pursue the project development by including these unnecessary opcodes.

There are two major concerns in implementing Java ISA – the instruction set itself and the JVM stack machine (as explained in Chapter 2, Stack Computers). Preliminarily, only the basic opcodes will be implemented, including all stack related, arithmetic, logic and return/jump operations but ruling out the remaining such as long, float, double, array and conversion operations. The instructions being implemented in MJava project is show in **List 1** (next page).

Pushing Constants onto the Stack

bipush	*sipush*

Loading Local Variables onto the Stack

iload	*iload_<n>*

Storing Stack Values into Local Variables

iconst_<n>	*istore_<n>*
istore	*iinc*

Stack Instructions

nop	*dup*
pop	*dup2*
pop2	*swap*

Arithmetic Instructions

iadd	*ineg*
isub	

Logical Instructions

ishl	*ior*
ishr	*ixor*
iand	

Control Transfer Instructions

if_icmpeq	*if_icmpge*
if_icmpne	*goto*
if_icmplt	*jsr*
if_icmpgt	*ret*
if_icmple	

List 1: JVM instructions being implemented in MJava

3.2 DEVELOPMENT AND SIMULATION

Development of the project were approached by systematical individual approach. The design was subjected to a work breakdown system (WBS) of a full integrated processor system. Necessary modules are identified and approach individually – ALU, stacks, program counter and data path. With the individual approach, each module were able to be subjected to several test simulations. These modules were then integrated using the data path design, done in behavioural style and tested again as a whole unit. This can ensure that the integrity in whole and reliability of each module is proven.

Simulation of the processor can be done in one of two ways or both combined, of Verilog HDL model and/or block diagram schematics. While Verilog HDL model is a text-based approach, block diagram schematics is a graphical-based approach that seems appropriate and easier option for simple and fundamental operations. However, when designing a far more complex processor, it is best to choose to model in Verilog. Simulation and synthesis will go through two procedures of functional simulation and timing simulation. The former only concerns of its fundamental of functional operation, while the latter takes into account additional parameter – processor clock.

3.2.1 Using Behavioural Verilog

It was decided that the design of the entire project would done in behavioural Verilog. The behavioural programming is similar to programming in C and C++, allowing designers to define their circuits based on how it would behave or function. This is contrast to RTL coding style that define components of circuits and their connections. With behavioural style, the code is more transparent, portable and extensible even to other people who decided to proceed the project works.

3.2.2 Using Extensive Test bench

In this project, some glitches resulted in possibility of no hardware

implementation for verification. Thus, to verify that the design works, an extensive testing fixture must apply. Simulations were to run with strict rules, experimenting with every possible corner case – reaching the limit of what the modules can do and go beyond it.

3.3 HARDWARE VERIFICATION

When designing the microprocessor, the targeted device must be kept in mind. Most times, a circuit design for a particular device are not synthesizable on other device. Although the codes are written in portable behavioural style and the simulations shows expected execution.

It is highly preferred to verify circuit design with hardware implementation. But circuit synthesis can be an issue. Early in the project progression, it has been decided the design will be implemented in Altera's FPGA development kit, but halfway through, it was switched to Xilinx's FPGA due to limitation in Quartus II compiler.

4. RESULTS & DISCUSSION

4.1 ARITHMETIC & LOGIC UNIT

The Arithmetic & Logic Unit (ALU) was design as a 32-bit signed two's complement arithmetic and logic evaluator. The inputs into the module consists of two input arguments, which are to be evaluated, and an instructions selector. The output from the modules are the evaluation resultants, embodied with three status flags – Z flag for indicating zero value resultant, V flag for indicating an overflow and N flag for indicating the sign of the resultant. **Table 4.1** shows the relevant ports declared inside the ALU module.

The status flags were designed from scratch, although the two Z and N flags are very simple. Z flag indicate a zero value resultant, achieved by ANDing the resultant bits. Z flag is set to one (1) if the resultant in zero in value and reset to zero (0) if it is a non zero value. N flag indicate the sign of the resultant, and thus only taking the most significant bit (MSB) of the resultant into argument. N flag is set to one (1) if the resultant is a negative number and reset to zero (0) if it is positive. V flag has more complex design, where it has to indicate whether an overflow had occur while evaluating the input arguments. This usually can occur with the following situations,

- Two positive values added.
- two negative values subtracted.
- Two values (of any sign) multiplied.

V flag was design by applying Karnaugh Map and supplying the above situation. V flag is set to one (1) if an overflow occur and reset to zero (0) if not. **Figure 4.1** presents the Verilog equation used to define these flags.

```
assign flag_z = result? 0 : 1;
assign flag_n = result[lop-1];
assign flag_v =   (instr==2'b01)?
      ((A[lop-1] ^ result[lop-1]) &
      (B[lop-1] ^ result[lop-1])) :
                  ((instr==2'b10)?
      ((A[lop-1] ^ result[lop-1]) &
      (B[lop-1] ~^ result[lop-1])) :
                  1'b0);
```

Figure 4.1: Status flag defined in ALU module

Table 4.1: Ports in MJava ALU module

Ports	Type	Width	Description
A	input	32	First argument of the evaluation.
B	input	32	Second argument of the evaluation.
instr	input	8	Select operation to perform. Also act as a trigger to invoke operation selection.
Cout	output	1	The 33rd bit, reserved for future use.
result	output	32	The resultant of the ALU evaluation.
flag_z	output	1	Asserted when the resultant is zero
flag_v	output	1	Asserted when a an overflow occur
flag_n	output	1	Asserted when the resultant is a negative number.

Input `instr` is fetched directly from the opcode itself. This should behave like a switch, where the module will be asserted when the input instr is assigned with a valid opcode from the *bytecodes* stream. Operations are chosen with a case statement, putting the input instr into the case argument. A total of 27 instructions available for execution, with highly extensible data path. The instructions chosen are fundamentals and significant to ensure reliability of the processor. **Table 4.2** shows list of instructions available in the ALU module.

Table 4.2: Instructions executed within ALU module

Instruction	Imp.	Description
nop	•	No operation.
iadd	•	Add two int operands. Two values popped from stack.
isub	•	Subtract two int operands. Two values popped from stack.
ineg	•	Negate an int operand. One value popped.
ishl	•	Arithmetic shift left. Two values popped.
ishr	•	Arithmetic shift right. Two values popped.
iand	•	Boolean AND two int.
ior	•	Boolean OR two int.
ixor	•	Boolean XOR two int.

- *Note: Imp. = implementation.*

At the moment the implementation status shows only limited instruction had been implemented. The ALU module is designed to use take operations selection arguments directly from the opcodes for high extensibility. Any instruction that put two values into argument with one resultant can be easily implemented inside the module. Full Verilog code and simulation result for the ALU module can be referred in **Appendix B1**.

4.2 OPERANDS AND RETURN STACKS

The operands and return stacks are instantiated from the same LIFO stack module design. However, instead of having a single stack for operands and return addresses, they are separated to increase integrity in performing nested subroutines and prevent mismatch fetch of operands for operation. It would give a great complexity if the operands and return addresses were to share same stack, resulting in an inefficient and larger-size cores.

Stack operates in two modes; (i) *push* operand onto top most location and (ii) *pop* operand(s) from the one or two top most location(s). Any data *pushed*

and *popped* from the stack is of 32-bit width. Prior to *push* operation, smaller data types are signed extended, while larger data types are broken into several 32-bit width data. *Pop* operation will output a 32-bit wide data. It is up to the central processing to combine or disjoint the necessary operands. For the *pop* operation, a single *pop* will read the top of stack and write to the output port 1. A double *pop* will read the top two of stack and write to the output port 1 and port 2.

The module design utilise hardware memory array for the stack, declared as type `reg`. It can occupy up to eight (8) data of 32-bit width, stored in systematic bottom-top fills. Stack pointer indicates where data input will be stored, starting at bottom most location and increase by a location after each successful *push* and decrease by a location after each successful *pop*. The memory array also behave like a circular buffer. It rotates to the top most location whenever it reaches lower than the bottom most location and rotates to the bottom most location whenever it reached upper that the top most location.

As in the ALU module, input `instr[1:0]` act as a trigger to execute the selected operation, where it must be reset if not in use. The instruction value is fetched during the decode phase in the data path. An output `stStore` provides indication whenever data has successfully been *pushed*, assisting the data path to determine the appropriate next operation **Table 4.3** shows relevant ports declared inside the stack module.

The design approach maintain the stacks safe from data corruption due to manual overrides in input ports. The stacks remain inside the core without direct connections and accessed only via double doors system, where instructions instr are not direct association of any opcodes – unlike the ALU. **Figure 4.2** shows how the memory array was declared. Full Verilog code and simulation results can be referred in **Appendix B2**.

Table 4.3: Ports in MJava stack module

Ports	Type	Width	Description
clk	input	1	Clock.
reset_n	input	1	Reset port
data_in	input	32	Data input (for push) port.
read_n	input	1	Enable read (pop) port.
write_n	input	1	Enable write (push) port.
pop_2	input	1	Double pop indicator.
data_out1	output	32	Output port 1.
data_out2	output	32	Output port 2.
pushed	output	1	Successful push indicator.
popped	output	1	Successful pop indicator.

```
module MStack(clk, data, instr, stStore, out);
parameter dep=8, spdep=3, lop=32;

input clk;
input [1:0] instr;
input `Lop data;
output stStore;
output `Lop out;

reg `Lop stackmem [dep-1:0];
reg [spdep-1:0] sptr;
reg stStore;
reg `Lop out;
```

FIgure 4.2: MJava stack module declaration. reg type stackmem[7:0] is the actual stack memory array.

4.3 PROGRAM COUNTER

Program Counter (PC) is also a stack-based module, but instead, utilises a FIFO type stack. The purpose of PC is mainly to provide a storage to streams of instruction like a long buffer. Thus it allows *bytecodes* stream to be kept in close to the processor cores. The implementation of FIFO-type PC also add the extensibility to perform branch and jump instructions.

The design is fairly simple and common. It has five (5) input port and four (4) output port. **Table 4.4** shows the relevant ports declared inside the PC module. The PC has a memory array `pc_mem[]` that stores all the instructions, in bytes. The memory array has 16 locations of a byte wide. The small size is chosen as experimental value. It is easily extensible with only a line of code change. Since PC is FIFO stack, it has two pointers – read and write. These pointers indicate the read and write location within the `pc_mem[]` array. Whenever a buffer overflow or underflow occur, a flag is asserted at the output port (see **Table 4.4**). An internal counter is used to determine whether or not overflow or underflow occur.

This module start with writing instructions, whenever `write` port is asserted, from external programmer, buffering them into the memory array. During this period, no operation is allowed in the data path and read operation remain de-asserted. As soon as the `write` port get de-asserted, it indicate to the data path that it is ready for processor operations. Succeeding operation (read from PC) is controlled by the data path, until interrupted again whenever `write` port is reasserted. The cycle continues.

Table 4.4: Ports in MJava program counter module

Ports	Type	Width	Description
`clk`	input	1	Clock.
`reset_n`	input	1	Reset port
`data_in`	input	32	Data input (for push) port.
`read_n`	input	1	Enable read (pop) port.

Ports	Type	Width	Description
write_n	input	1	Enable write (push) port.
data_out	output	32	Output port.
full	output	1	PC overflow indicator.
empty	output	1	PC underflow indicator.
half	output	1	Indicate pointer at midway.

4.4 DATAPATH AND MODULES INTEGRATION

Datapath module is a collections of wires and ports connecting the necessary external modules to their respective operation. Datapath is responsible for the integration between modules instantiated. It provides the way for the ALU, operands and return stacks, and program counter to function as a single unit. Design technique employed in the project is simple, but as number of instructions increase, it also increase in complexity. As in other modules, the data path was developed using behavioural style Verilog.

Datapath module has three modules instantiation - the ALU and operand stack and program counter. The data path utilises many `always@` block, triggering action only when certain input changes values. PC fetches *bytecodes* stream by bytes to the data path, whenever `pc_read` is asserted. It then decode the opcode fetched and translated it for proper parameters setting in the first `always` block. Following through the sequence, the opcode parameters will indicate which modules to assert first and whether to use the stacks, local variables, etc. Operations in the data path are of sequential flow. The basic operation sequence is presented in **Figure 4.5**.

Since microprocessor circuits are meant to execute concurrently, design in sequential flow resulted in a mixed complexity. Nevertheless, the performance were not taxed since the complexity only lies on the codes and not the circuitry. The simulation runs several instructions and testing the

functionality of each modules. Instructions fetched are shown in **Figure 4.4**, where immediate values were *pushed* several times onto the stack before calling the addition, subtraction, negate and swap operations.

Datapath has four (4) input port and one (1) output port. Its significant input argument is `byte_in[]`, used to transfer instructions from external programmer and buffer them inside the PC module. The `byte_in[]` is of a byte wide, which correspond to the PC byte wide input, storage and output. A master `reset` port, is used to reset and reinitialized all inputs and pointers. **Table 4.5** shows the relevant ports declared inside the MJava main data path module.

Decoding instructions required several `always@` statements that get asserted whenever the input arguments changes values. As a result, many `regs` and `wires` are declared along with the inout ports to assist the decoding operations. **Figure 4.3** shows the MJava Datapath module declaration. Full code of the MJava Datapath module can be referred in **Appendix B4**.

Table 4.5: Ports in MJava Datapath module

Ports	Type	Width	Description
`clk`	input	1	Clock.
`reset`	input	32	Manual reset.
`ctrlword`	input	2	Fetch the bytecodes.
`result`	output	1	Output to external.

```
module MJava(clk, reset, write, byte_in, out_stream);

      input                          clk;
      input                          reset;
      input                          write;
      input      [`BYTE_WIDTH-1:0]   byte_in;

      output     [`INT_WIDTH-1:0]    out_stream;

      wire                           clk;
      wire                           reset;
      wire                           write;
      wire       [`BYTE_WIDTH-1:0]   byte_in;

      reg        [`INT_WIDTH-1:0]    out_stream;
```

Figure 4.3: MJava datapath module declaration. Many type of regs and wires were declared and used.

```
always @(posedge clk) begin
            write = 1;
            byte_in = 8'h10;  // push byte
      #`PER byte_in = 8'hAA;
      #`PER byte_in = 8'h10;  // push byte
      #`PER byte_in = 8'hBB;
      #`PER byte_in = 8'h11;  // push short
      #`PER byte_in = 8'hCC;
      #`PER byte_in = 8'hDD;
      #`PER byte_in = 8'h60;  // integer add
      #`PER byte_in = 8'h78;  // integer shl
      #`PER byte_in = 8'h80;  // integer or
      #`PER byte_in = 8'h3c;  // istore_1
      #`PER byte_in = 8'h1b;  // iload_1
      #`PER byte_in = 8'h04;  // iconst_2
      #`PER byte_in = 8'h00;
      #`PER byte_in = 8'h00;
            write = 0;
      #500 $stop;
end   // `PER == 10;
```

Figure 4.4: Lines of code fetched for testing datapath functionality.

Table 4.6: Instructions implemented in MJava processor.

Instruction	Imp.	Description
nop	•	No operation.
iadd	•	Add two int operands. Two values popped from stack.
isub	•	Subtract two int operands. Two values popped from stack.
ineg	•	Negate an int operand. One value popped.
ishl	•	Arithmetic shift left. Two values popped.
ishr	•	Arithmetic shift right. Two values popped.
iand	•	Boolean AND two int.
ior	•	Boolean OR two int.
ixor	•	Boolean XOR two int.
bipush	•	An immediate byte is pushed onto the operand stack.
sipush	•	An immediate short is pushed onto the operand stack.
swap	•	The top two value in operand stack are swapped and pushed back onto the stack.
istore_<n>	•	Store value from operand stack into local variable of corresponding <*n*>
iload_<n>	•	Load value from local variable of corresponding <*n*> and push onto top of operand stack.
iconst_<n>	•	Pushing constants of corresponding <*n*> onto the operand stack.
if_icmp<cond>		Branch if int comparison succeeds. Two-byte jump address is embodied in the instruction stream. Two values popped from the stack, where value1 is top of stack and value2 is next top of stack.
if_icmpeq		succeeds if and only if *value1* = *value2*
if_icmpne		succeeds if and only if *value1* ≠ *value2*
if_icmplt		succeeds if and only if *value1* < *value2*
if_icmple		succeeds if and only if *value1* ≤ *value2*
if_icmpgt		succeeds if and only if *value1* > *value2*
if_icmpge		succeeds if and only if *value1* ≥ *value2*

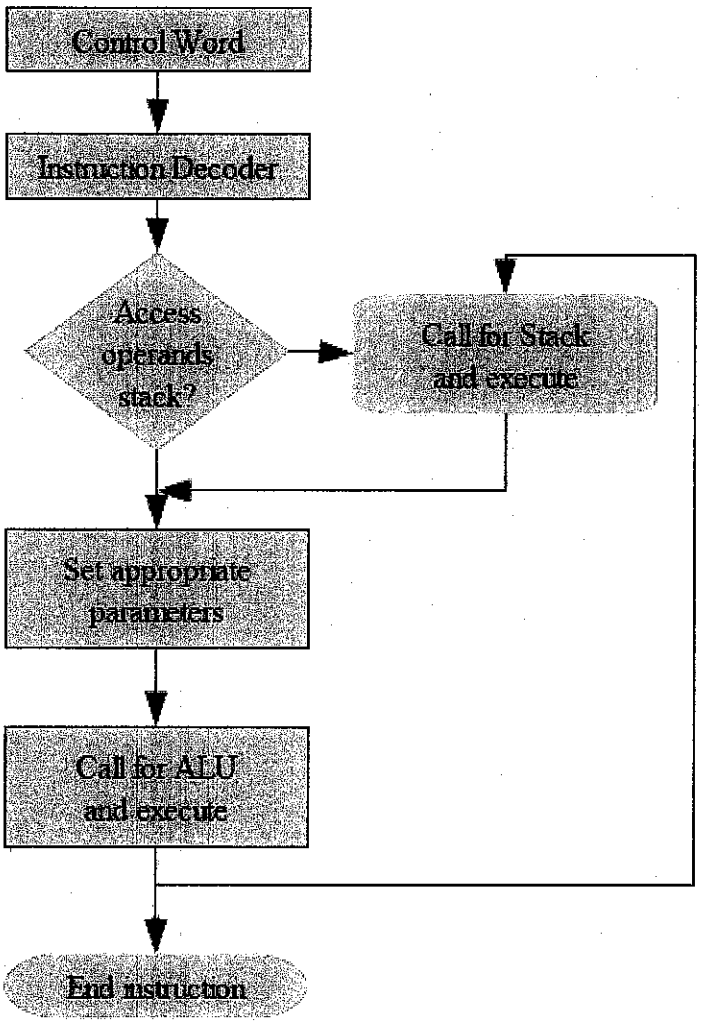


Figure 4.5: Sequential flow of MJava data path

5. CONCLUSION & RECOMMENDATION

Project title Java-based Microprocessor is a huge topic by itself. However, with proper planning and specific target, it did not appear to be as overwhelming as some people would assume. Throughout the project several constraints and obstacles faced that in some ways change the direction. Nevertheless, the project manage to achieve its basic objective of implementing the core of JVM into a hardware circuitry.

The ALU module was developed accordingly, achieving its target as computation module for arithmetic and logical operations. All necessary ALU instructions had been implemented but with the lack of more complex synthesis.

The stacks module, the operand stack and program counter is most convincing fully synthesizable modules. Their exceptions lies on proper design from highly reliable sources, proven and reused many times by others. Circuit synthesis are presented in **Appendix C**.

The Datapath module achieve its purpose, but lack of understanding in data path design led to lengthy HDL code. It meets the objectives of linking other modules and allow them to work as unit and allow further extension of additional instructions easily without tempering with original design. Decision to develop the data path using comprehensible behavioural style coding proves to be advantageous.

Performance may not be the strong side of this project, yet it is a pilot project for other colleagues to pursue in the future. The implementation of JVM instructions are limited to basic operations involving only integers, shorts and bytes. Although the data path design was unique, there is room for improvement especially in term of pipelining.

6. REFERENCES

[1] Harlan McGhan and Mike O'Connor, *picoJava: A Direct Execution Engine for Java Bytecode,* Sun Microsystems

[2] Dr. Andreas Steininger and Dr. Peter Puschner, *Java Optimized Processor,* 2005

[3] Bernd Paysan, *b16 – a Forth Processor in FPGA,* 2003

[4] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification,* 2nd Edition, Addison-Wesley

[5] *The Java Virtual Machine Specification,* Sun Microsystems, 1998

[6] Philip Koopman Jr., *Stack Computers: The New Wave*, Mountain View Press, 1989

[7] Carpinelli, *Computer Systems Organization & Architecture*

[8] Mark Gordon Arnold, *Verilog Digital Computer Design: Algorithms to Hardware,* Prentice Hall PTR

[9] Weng Fook Lee, *Verilog Coding for Logic Synthesis,* Wiley-Interscience

7. APPENDICES

Appendix A1: MJava ALU Verilog Code

Appendix A2: MJava Stack Verilog Code

Appendix A3: MJava Program Counter Verilog Code

Appendix A4: MJava Datapath Verilog Code

Appendix B1: MJava ALU Simulation Results

Appendix B2: MJava Stack Simulation Results

Appendix B3: MJava Program Counter Simulation Results

Appendix B4: MJava Datapath Simulation Results

Appendix C: Stacks Synthesized Circuits

Appendix D1: JVM Instructions Hexadecimal Values

Appendix D2: JVM Instructions and Operands Description

APPENDIX A1: MJAVA ALU VERILOG CODE

```
`define Lop [lop-1:0]
`define Loc [loc-1:0]
`timescale 1ns / 1ns

module MALU(A, B, instr, Cout, result, flag_z, flag_v, flag_n);
parameter lop=32, loc=8;

   input `Lop A, B;
   input `Loc instr;
   output `Lop result;
   output Cout, flag_z, flag_v, flag_n;

   reg   `Lop result;
   reg   Cout;
   wire  flag_z, flag_n, flag_v;

   always @(A or B or instr)
   begin
      case(instr)
         8'h84:   {Cout, result} = (A + B);      // increment (need touch up)
         8'h60:   {Cout, result} = (A + B);      // addition
         8'h64:   {Cout, result} = (A - B);      // subtraction
         8'h74:   {Cout, result} = (~A + 1'b1); // negation
         8'h78:   {result} = (A << B);     // shift left
         8'h7a:   {result} = (A >> B);     // shift right
         8'h7e:   {result} = (A & B);      // boolean AND
         8'h80:   {result} = (A | B);      // boolean OR
         8'h82:   {result} = (A ^ B);      // boolean XOR
      endcase
   end

   assign flag_z = result? 0 : 1;
   assign flag_n = result[lop-1];
   assign flag_v = (instr==2'b01)?
                     ((A[lop-1] ^ result[lop-1]) & (B[lop-1] ^ result[lop-1])) :
                        ((instr==2'b10)?
                        ((A[lop-1] ^ result[lop-1]) & (B[lop-1] ~^ result[lop-1])) :
                     1'b0);
endmodule // alu
```

APPENDIX A2: MJAVA STACKS VERILOG CODE

```
`timescale 1ns / 1ns

// DEFINES
`define  DEL   1            // Clock-to-output delay. Zero
                            // time delays can be confusing
                            // and sometimes cause problems.
`define  ST_DEPTH 8         // Depth of stack (number of bytes)
`define  ST_BITS  3         // Number of bits required to
                            // represent the FIFO size
`define  INT_WIDTH   32     // Width of stack data

module MStack(
         clock,
         reset_n,
         data_in,
         read_n,
         write_n,
         pop_2,
         data_out1,
         data_out2,
         pushed,
         popped);

// INPUTS
input                       clock;
input                       reset_n;
input [`INT_WIDTH-1:0]      data_in;
input                       read_n, write_n;
input                       pop_2;

// OUTPUTS
output    [`INT_WIDTH-1:0]  data_out1, data_out2;
output                      pushed;
output                      popped;

// SIGNALS DECLARATIONS
wire                        clock;
wire                        reset_n;
wire      [`INT_WIDTH-1:0]  data_in;
wire                        read_n, write_n;
wire                        pop_2;
reg       [`INT_WIDTH-1:0]  data_out1, data_out2;
reg                         pushed;
reg                         popped;

reg       [`INT_WIDTH-1:0]  st_mem[`ST_DEPTH-1:0];
                            // How many locations in the stack
                            // are occupied?
reg    [`ST_BITS-1:0]       st_pointer;


// ASSIGN STATEMENTS

// MAIN CODE

// Look at the edges of reset_n
always @(reset_n) begin
   if (reset_n == 1'b1) begin
      // Reset the stack pointer
      #`DEL;
      assign st_pointer = `ST_DEPTH - 1'b1;
      assign popped = 0;
      assign pushed = 0;
   end
   else begin
      #`DEL;
      deassign st_pointer;
      deassign popped;
      deassign pushed;
   end
end

// Look at the rising edge of the clock
always @(posedge clock) begin
   // Popping data from stack
   if (read_n == 1'b1) begin
```

```
      // Output the data
      data_out1 = #`DEL st_mem[st_pointer];
      //st_mem[st_pointer] = 32'h00000000;
      // Decrement the stack pointer
      // If the pointer has gone beyond the bottom of stack,
      // bring it to the top of stack.
      if (st_pointer == 0)
         st_pointer = #`DEL `ST_BITS'b111;
      else
         st_pointer = #`DEL st_pointer - 1;

      if (pop_2 == 1'b1) begin
         data_out2 = #`DEL st_mem[st_pointer];
         //st_mem[st_pointer] = 32'h00000000;
         if (st_pointer == 0)
            st_pointer = #`DEL `ST_BITS'b111;
         else
            st_pointer = #`DEL st_pointer - 1;
      end
      popped = ~popped;
   end
   // Pushing data onto stack
   if (write_n == 1'b1) begin
      // Increment the stack pointer
      // If the pointer has gone beyond the top of stack,
      // bring it to the bottom of stack.
      if(st_pointer == `ST_DEPTH-1)
         st_pointer = #`DEL `ST_BITS'b0;
      else
         st_pointer = #`DEL st_pointer + 1;
      // Store the data
      st_mem[st_pointer] = #`DEL data_in;
      pushed = ~pushed;
   end
end
endmodule
```

APPENDIX A3: MJAVA PROGRAM COUNTER VERILOG CODE

```
`timescale 1ns / 1ns

// DEFINES
`define  DEL   1            // Clock-to-output delay. Zero
                            // time delays can be confusing
                            // and sometimes cause problems.
`define  PC_DEPTH 16     // Depth of PC (number of bytes)
`define  PC_HALF  8      // Half depth of PC
                            // (this avoids rounding errors)
`define  PC_BITS  4      // Number of bits required to
                            // represent the PC size
`define  BYTE_WIDTH  8      // Width of PC data

module MPC(
         clock,
         reset_n,
         data_in,
         read_n,
         write_n,
         data_out,
         full,
         empty,
         half);

// INPUTS
input                        clock;
input                        reset_n;
input [`BYTE_WIDTH-1:0]      data_in;
input                        read_n;
input                        write_n;

// OUTPUTS
output    [`BYTE_WIDTH-1:0] data_out;
output                       full;
output                       empty;
output                       half;

// SIGNALS DECLARATIONS
wire                         clock;
wire                         reset_n;
wire      [`BYTE_WIDTH-1:0] data_in;
wire                         read_n;
wire                         write_n;
reg       [`BYTE_WIDTH-1:0] data_out;
wire                         full;
wire                         empty;
wire                         half;

reg       [`BYTE_WIDTH-1:0] pc_mem[0:`PC_DEPTH-1];
                            // How many locations in the PC
                            // are occupied?

reg    [`PC_BITS-1:0]     counter;

reg    [`PC_BITS-1:0]     rd_pointer;

reg    [`PC_BITS-1:0]     wr_pointer;

// ASSIGN STATEMENTS
assign #`DEL full = (counter == `PC_DEPTH) ? 1'b1 : 1'b0;
assign #`DEL empty = (counter == 0) ? 1'b1 : 1'b0;
assign #`DEL half = (counter >= `PC_HALF) ? 1'b1 : 1'b0;

// Look at the edges of reset_n
always @(reset_n) begin
  if (reset_n == 1'b1) begin
     // Reset the PC pointer
     #`DEL;
     assign rd_pointer = `PC_BITS'b0;
     assign wr_pointer = `PC_BITS'b0;
     assign counter = `PC_BITS'b0;
  end
  else begin
     #`DEL;
     deassign rd_pointer;
     deassign wr_pointer;
```

```
      deassign counter;
   end
end

// Look at the rising edge of the clock
always @(posedge clock) begin
   if (read_n == 1'b1) begin
      // Check for PC underflow
      if (counter == 0) begin
         $display("\nERROR at time %0t:", $time);
         $display("PC Underflow\n");
         $stop;      // Use $stop for debugging
      end
      // If we are doing a simultaneous read and write,
      // there is no change to the counter
      if (write_n == 1'b0) begin
         // Decrement the PC counter
         counter <= #`DEL counter - 1;
      end
      // Output the data
      data_out <= #`DEL pc_mem[rd_pointer];

      // Increment the read pointer
      // Check if the read pointer has gone beyond the
      // depth of the PC. If so, set it back to the
      // beginning of the PC
      if (rd_pointer == `PC_DEPTH-1)
         rd_pointer <= #`DEL `PC_BITS'b0;
      else
         rd_pointer <= #`DEL rd_pointer + 1;
   end
   if (write_n == 1'b1) begin
      // Check for PC overflow
      if (counter >= `PC_DEPTH) begin
         $display("\nERROR at time %0t:", $time);
         $display("PC Overflow\n");

         // Use $stop for debugging
         $stop;
      end

      // If we are doing a simultaneous read and write,
      // there is no change to the counter
      if (read_n == 1'b0) begin
         // Increment the PC counter
         counter <= #`DEL counter + 1;
      end

      // Store the data
      pc_mem[wr_pointer] <= #`DEL data_in;

      // Increment the write pointer
      // Check if the write pointer has gone beyond the
      // depth of the PC. If so, set it back to the
      // beginning of the PC
      if (wr_pointer == `PC_DEPTH-1)
         wr_pointer <= #`DEL `PC_BITS'b0;
      else
         wr_pointer <= #`DEL wr_pointer + 1;
   end
end
endmodule   // PC
```

APPENDIX A4: MJAVA DATAPATH VERILOG CODE

```
`timescale 1ns / 1ns

//DEFINES
`define BYTE_WIDTH 8
`define INT_WIDTH 32
`define PER 10
`define DEL 1

module MJava(clk, reset, write, byte_in, out_stream);

// INPUTS
   input                          clk;
   input                          reset;
   input                          write;
   input     [`BYTE_WIDTH-1:0]    byte_in;

// OUTPUTS
   output    [`INT_WIDTH-1:0]     out_stream;

// SIGNALS DECLARATIONS
   wire                           clk;
   wire                           reset;
   wire                           write;
   wire      [`BYTE_WIDTH-1:0]    byte_in;

   reg       [`INT_WIDTH-1:0]     out_stream;

   wire      [`BYTE_WIDTH-1:0]    byte_out;
   wire                           full;
   wire                           empty;
   wire                           half;
   wire      [`INT_WIDTH-1:0]     st_out1;
   wire      [`INT_WIDTH-1:0]     st_out2;
   wire      [`INT_WIDTH-1:0]     aluResult;
   wire                           st_pushed;

   reg                            pc_read;
   reg       [`INT_WIDTH-1:0]     buffA;
   reg       [`INT_WIDTH-1:0]     buffB;
   reg       [`BYTE_WIDTH-1:0]    byte1;
   reg       [`BYTE_WIDTH-1:0]    byte2;
   reg       [`BYTE_WIDTH-1:0]    opcode;
   reg       [`BYTE_WIDTH-1:0]    aluOper;
   reg       [1:0]                counter_pc;
   reg       [1:0]                counter_op;
   reg                            counter_5f;
   reg       [1:0]                op_count;
   reg       [1:0]                clk_count;
   reg                            decode_n;
   reg                            opcode_n;
   reg                            operand_n;
   reg                            execute_n;
   reg                            execute_clk;
   reg                            st_read;
   reg                            st_write;
   reg                            pop_2;
   reg       [`INT_WIDTH-1:0]     local_var [0:4];


// ------------------------------------------------------
// Instantiating the necessary modules for the hardware
// configuration and their ports designations.
// ------------------------------------------------------
   MPC pcounter(
      .clock(clk),
      .reset_n(reset),
      .data_in(byte_in),
      .read_n(pc_read),
      .write_n(write),
      .data_out(byte_out),
      .full(full),
      .empty(empty),
      .half(half)
      );

   MStack opstack(
```

```
      .clock(clk),
      .reset_n(reset),
      .data_in(buffA),
      .read_n(st_read),
      .write_n(st_write),
      .pop_2(pop_2),
      .data_out1(st_out1),
      .data_out2(st_out2),
      .pushed(st_pushed),
      .popped(st_popped)
      );

   MALU arith(
      .A(buffA),
      .B(buffB),
      .instr(aluOper),
      .Cout(Cout),
      .result(aluResult),
      .flag_z(flag_z),
      .flag_v(flag_v),
      .flag_n(flag_n)
      );

// MAIN CODE
   always @(reset) begin
      pc_read <= 0;
      counter_pc <= 0;
      counter_op <= 0;
      clk_count <= 2;
      decode_n <= 0;
      execute_n <= 0;
      execute_clk <= 0;
      st_read <= 0;
      st_write <= 0;
      pop_2 <= 0;
   end

   always @(posedge clk) begin
      // Filling up PC or start decode instructions
      if(write)
         decode_n <= 0;
      else begin
         clk_count <= clk_count + 1;
         if(clk_count == 0)
            opcode <= 8'h00;
         if(clk_count == 3) begin
            counter_pc <= 0;
            counter_op <= 0;
            op_count <= 0;
            operand_n <= 0;
            decode_n <= 1;
            opcode_n <= 1;
            pc_read <= ~pc_read; // enable read from pc
         end
      end

      // Wussup
   end   // end of always@

   always @(execute_n or execute_clk) begin
      if(execute_n) begin
         case(opcode)
            8'h10: begin
               buffA <= byte1;
               st_write <= ~st_write;  // enable write to stack
            end
            8'h11: begin
               buffA <= {byte1, byte2};
               st_write <= ~st_write;
            end
         endcase
      end
   end

   always @(opcode) begin
      // Decode the instructions
```

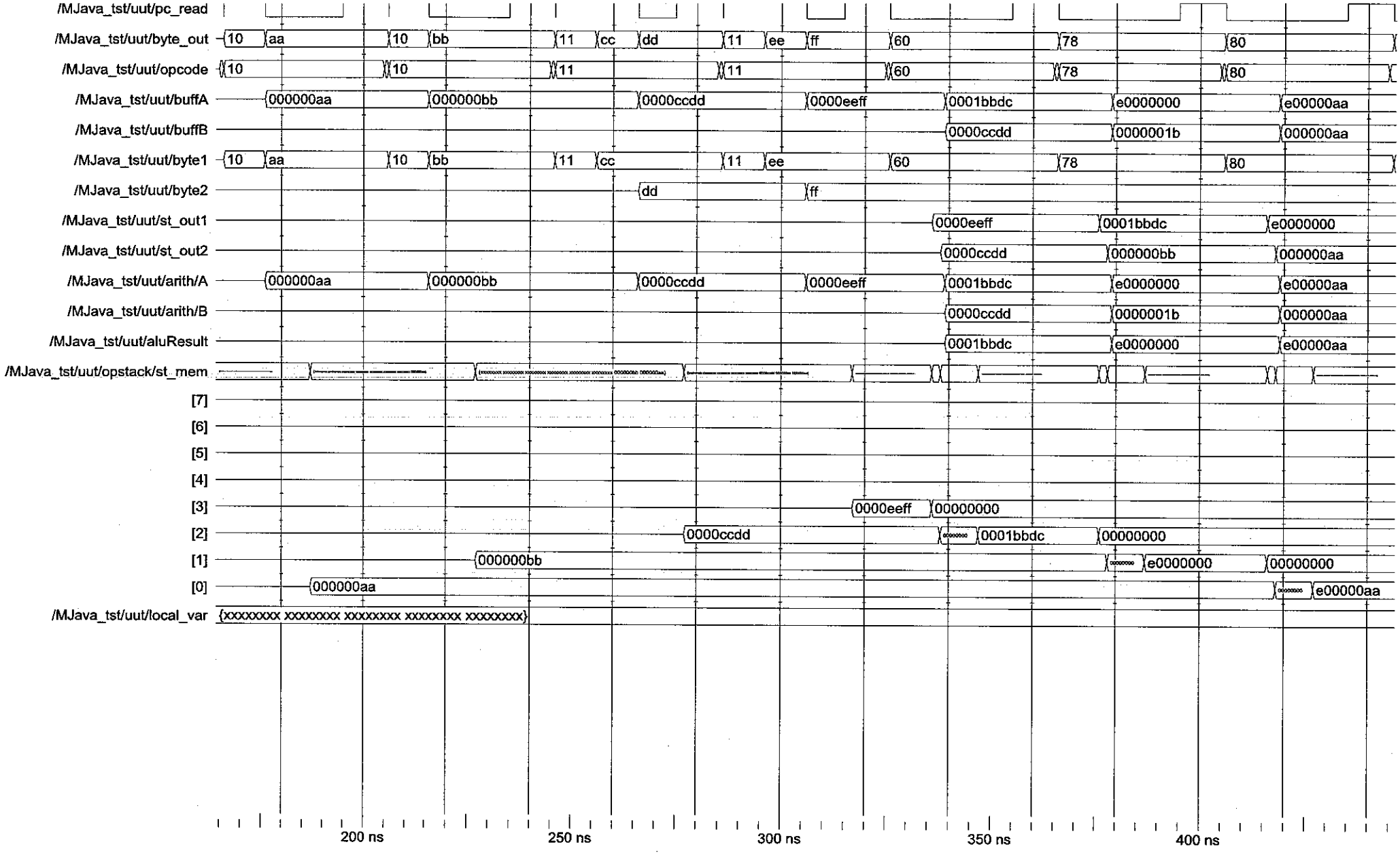
```
      case(opcode)
         8'h10: begin            // Case: bipush
            op_count = 1;
            counter_op = 0;
            pc_read = ~pc_read;  // enable read from pc
         end
         8'h11: begin
            op_count = 2;
            counter_op = 0;
            pc_read = ~pc_read;  // enable read from pc
         end
         8'h60: begin            // Case: iadd
            pop_2 <= ~pop_2;     // enable pop2
            st_read <= ~st_read; // enable read from stack
         end
         8'h64: begin            // Case: isub
            pop_2 <= ~pop_2;     // enable pop2
            st_read <= ~st_read; // enable read from stack
         end
         8'h74: begin            // Case: ineg
            st_read <= ~st_read;
         end
         8'h78: begin            // Case: ishl
            pop_2 <= ~pop_2;     // enable pop2
            st_read <= ~st_read; // enable read from stack
         end
         8'h7a: begin            // Case: ishr
            pop_2 <= ~pop_2;     // enable pop2
            st_read <= ~st_read; // enable read from stack
         end
         8'h7e: begin            // Case: iand
            pop_2 <= ~pop_2;     // enable pop2
            st_read <= ~st_read; // enable read from stack
         end
         8'h80: begin            // Case: ior
            pop_2 <= ~pop_2;     // enable pop2
            st_read <= ~st_read; // enable read from stack
         end
         8'h82: begin            // Case: ixor
            pop_2 <= ~pop_2;     // enable pop2
            st_read <= ~st_read; // enable read from stack
         end
         8'h5f: begin            // Case: swap
            pop_2 <= ~pop_2;     // enable pop2
            st_read <= ~st_read; // enable read from stack
         end
         8'h3b: begin            // Case: istore_0
            st_read <= ~st_read; // enable read from stack
         end
         8'h3c: begin            // Case: istore_1
            st_read <= ~st_read; // enable read from stack
         end
         8'h3d: begin            // Case: istore_2
            st_read <= ~st_read; // enable read from stack
         end
         8'h3e: begin            // Case: istore_3
            st_read <= ~st_read; // enable read from stack
         end
         8'h02: begin            // Case: iconst_m1
            buffA <= `INT_WIDTH'hFFFFFFFF;
            st_write <= ~st_write;
         end
         8'h03: begin            // Case: iconst_0
            buffA <= `INT_WIDTH'h00000000;
            st_write <= ~st_write;
         end
         8'h04: begin            // Case: iconst_1
            buffA <= `INT_WIDTH'h00000001;
            st_write <= ~st_write;
         end
         8'h05: begin            // Case: iconst_2
            buffA <= `INT_WIDTH'h00000002;
            st_write <= ~st_write;
         end
         8'h06: begin            // Case: iconst_3
            buffA <= `INT_WIDTH'h00000003;
```

```
            st_write <= ~st_write;
         end
         8'h07: begin            // Case: iconst_4
            buffA <= `INT_WIDTH'h00000004;
            st_write <= ~st_write;
         end
         8'h08: begin            // Case: iconst_5
            buffA <= `INT_WIDTH'h00000005;
            st_write <= ~st_write;
         end
         8'h1a: begin            // Case: iload_0
            buffA <= local_var[0];
            st_write <= ~st_write;
         end
         8'h1b: begin            // Case: iload_1
            buffA <= local_var[1];
            st_write <= ~st_write;
         end
         8'h1c: begin            // Case: iload_2
            buffA <= local_var[2];
            st_write <= ~st_write;
         end
         8'h1d: begin            // Case: iload_3
            buffA <= local_var[3];
            st_write <= ~st_write;
         end
      endcase
   end

   always @(byte_out) begin
      counter_op = counter_op + 1;
      if(opcode_n) begin
         opcode = byte_out;
         opcode_n = ~opcode_n;   // opcode is assigned
         pc_read = ~pc_read;     // disable read from pc
      end
      if(counter_op == 1)
         byte1 = byte_out;
      if(counter_op == 2)
         byte2 = byte_out;
      if(counter_op == op_count) begin
         pc_read = ~pc_read;     // disable read from pc
                                 // for operands retrieval
         execute_n = ~execute_n; // enable for EX stage
      end
   end

   always @(st_pushed) begin
      if(decode_n) st_write = ~st_write;
      case(opcode)
         8'h10: begin
            execute_clk <= ~execute_clk;
            execute_n <= ~execute_n;   // end of EX stage
         end
         8'h11: begin
            execute_clk <= ~execute_clk;
            execute_n <= ~execute_n;   // end of EX stage
         end
         8'h5f: begin
            if(~counter_5f) begin
               buffA <= st_out2;
               st_write = ~st_write;      // enable write to stack
               counter_5f <= counter_5f + 1;
            end
         end
      endcase
   end

   always @(st_popped) begin
      case(opcode)
         8'h60: begin               // iadd
            pop_2 <= ~pop_2;        // disable pop2
            st_read <= ~st_read;    // disable read from stack
            buffA <= st_out1;
            buffB <= st_out2;
            aluOper <= opcode;
```

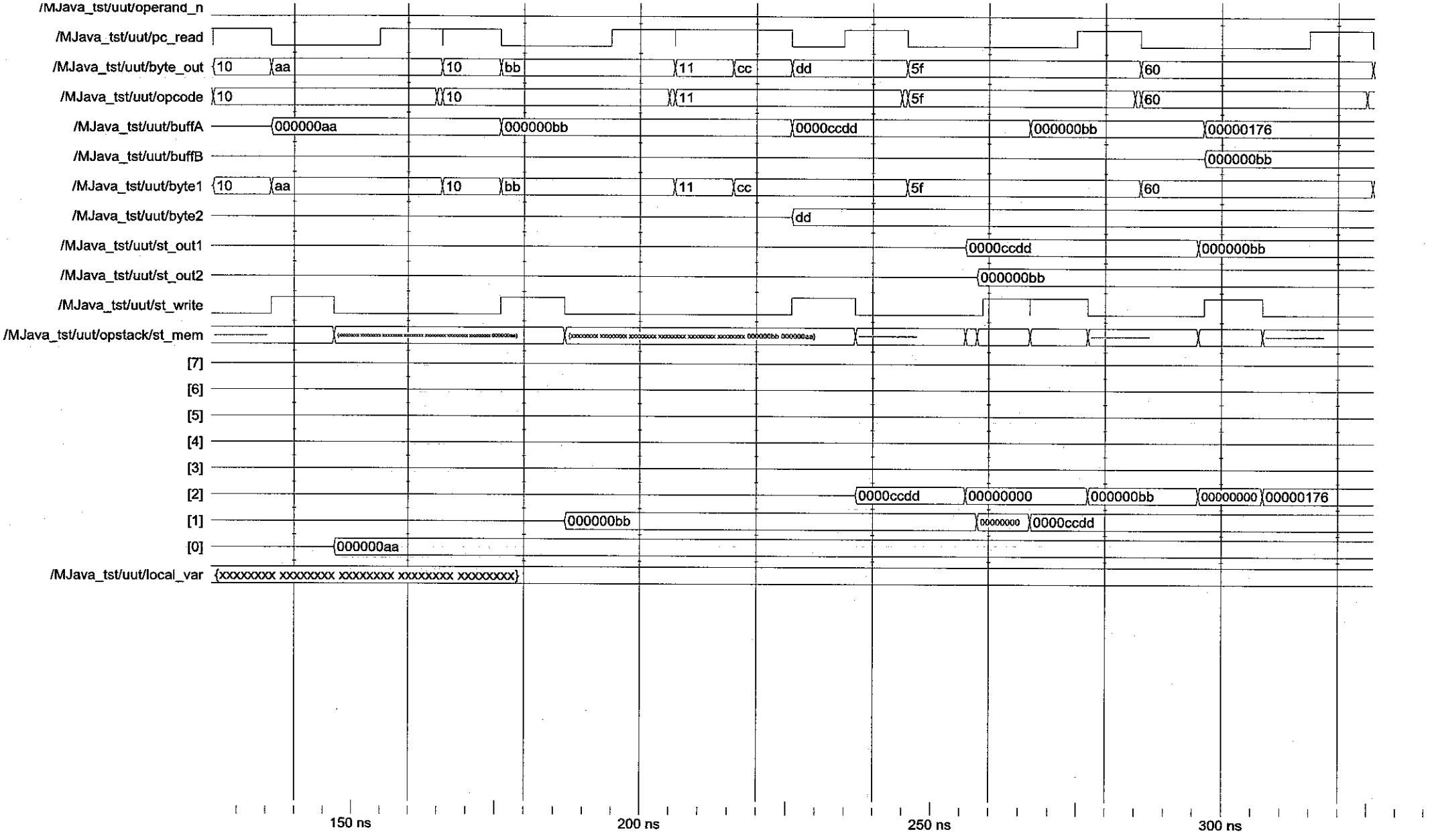
```
        end
        8'h64: begin                // isub
           pop_2 <= ~pop_2;         // disable pop2
           st_read <= ~st_read;     // disable read from stack
           buffA <= st_out1;
           buffB <= st_out2;
           aluOper <= opcode;
        end
        8'h74: begin                // ineg
           st_read <= ~st_read;     // disable read from stack
           buffA <= st_out1;
           aluOper <= opcode;
        end
        8'h78: begin                // ishl
           pop_2 <= ~pop_2;         // disable pop2
           st_read <= ~st_read;     // disable read from stack
           buffA <= st_out1;
           buffB <= {  st_out2[4],
                       st_out2[3],
                       st_out2[2],
                       st_out2[1],
                       st_out2[0]};   // select 5 LSB
           aluOper <= opcode;
        end
        8'h7a: begin                // ishr
           pop_2 <= ~pop_2;         // disable pop2
           st_read <= ~st_read;     // disable read from stack
           buffA <= st_out1;
           buffB <= {  st_out2[4],
                       st_out2[3],
                       st_out2[2],
                       st_out2[1],
                       st_out2[0]};   // select 5 LSB
           aluOper <= opcode;
        end
        8'h7e: begin                // iand
           pop_2 <= ~pop_2;         // disable pop2
           st_read <= ~st_read;     // disable read from stack
           buffA <= st_out1;
           buffB <= st_out2;
           aluOper <= opcode;
        end
        8'h80: begin                // ior
           pop_2 <= ~pop_2;         // disable pop2
           st_read <= ~st_read;     // disable read from stack
           buffA <= st_out1;
           buffB <= st_out2;
           aluOper <= opcode;
        end
        8'h82: begin                // ixor
           pop_2 <= ~pop_2;         // disable pop2
           st_read <= ~st_read;     // disable read from stack
           buffA <= st_out1;
           buffB <= st_out2;
           aluOper <= opcode;
        end
        8'h5f: begin                  // swap
           buffA <= st_out1;
           st_read <= ~st_read;       // disable read from stack
           st_write <= ~st_write;     // enable write to stack
           counter_5f <= 0;           // reset swap counter
        end
        8'h3b: begin                  // istore_0
           st_read <= ~st_read;       // disable read from stack
           local_var[0] <= st_out1;
        end
        8'h3c: begin                  // istore_1
           st_read <= ~st_read;       // disable read from stack
           local_var[1] <= st_out1;
        end
        8'h3d: begin                  // istore_2
           st_read <= ~st_read;       // disable read from stack
           local_var[2] <= st_out1;
        end
        8'h3e: begin                  // istore_3
           st_read <= ~st_read;       // disable read from stack
```

```
            local_var[3] <= st_out1;
         end
      endcase
   end

   always @(aluResult) begin
      buffA <= aluResult;
      aluOper <= 8'h00;
      st_write <= ~st_write;      // enable write to stack
   end
endmodule
```

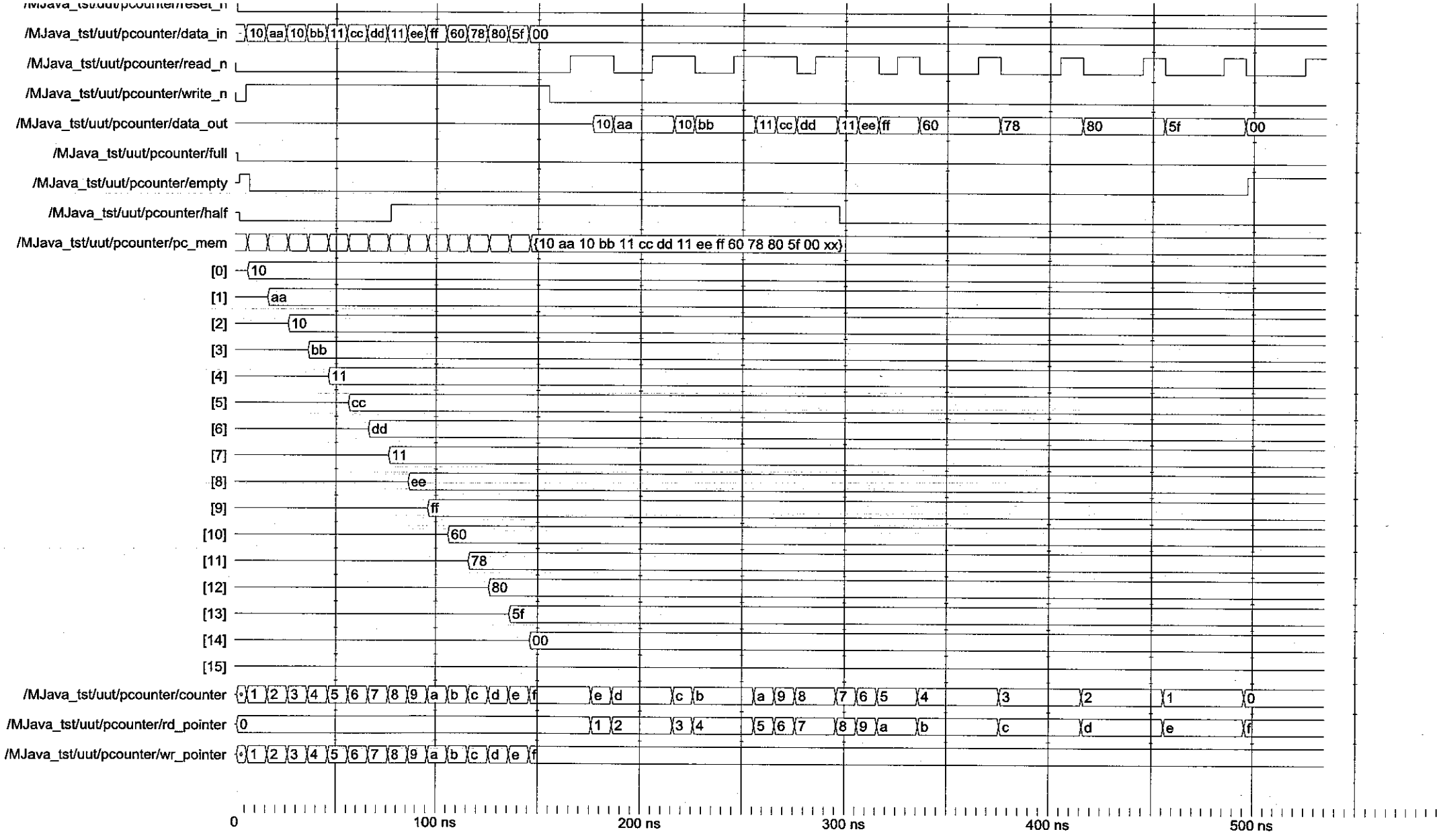
APPENDIX B: MJAVA SIMULATION RESULTS



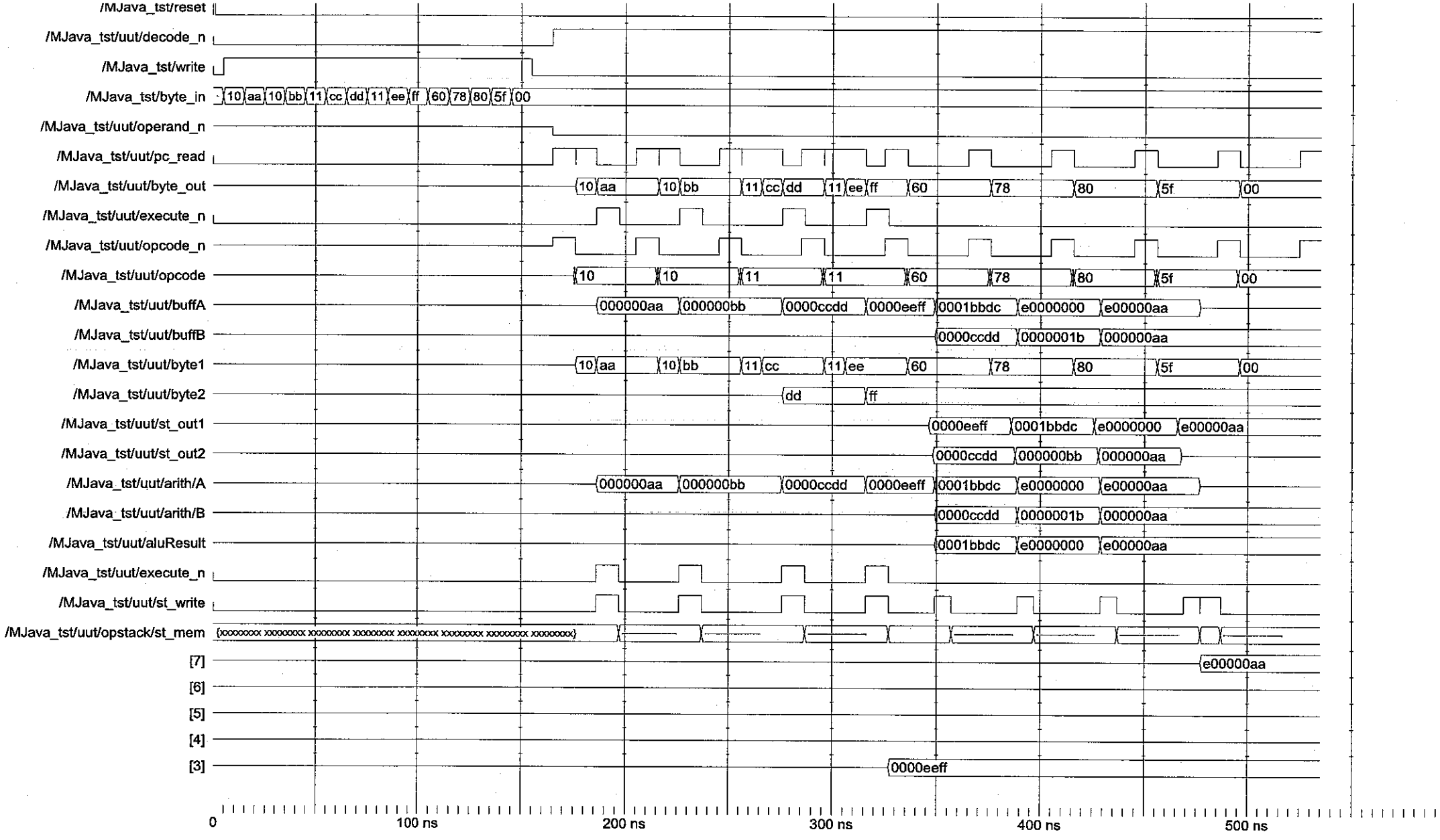
Entity:MJava_tst Architecture: Date: Wed May 10 11:39:03 Malay Peninsula Standard Time 2006 Row: 1 Page: 1



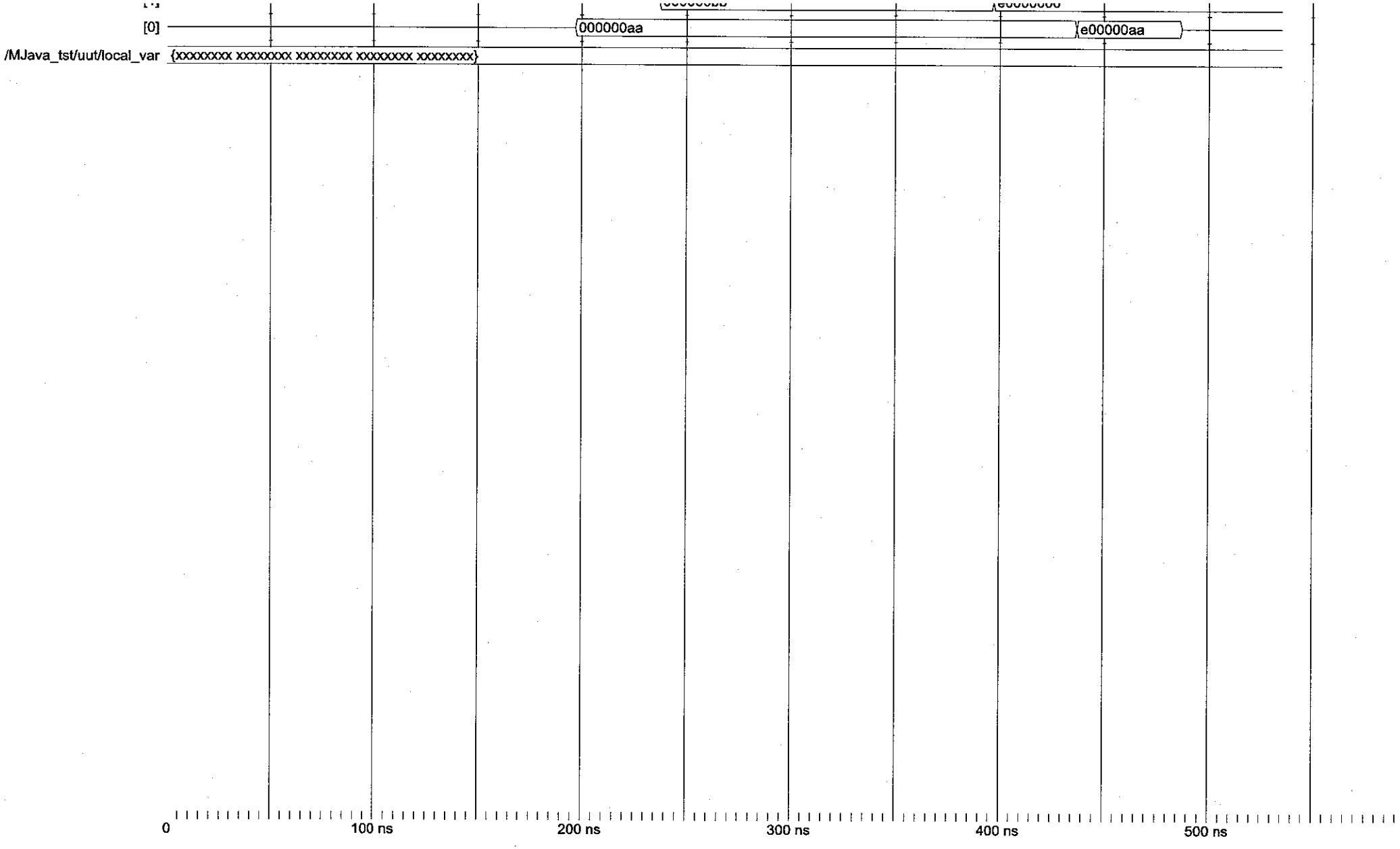
Entity:MJava_tst Architecture: Date: Wed May 10 11:33:46 Malay Peninsula Standard Time 2006 Row: 1 Page: 1



Entity:MJava_tst Architecture: Date: Wed May 10 12:43:20 Malay Peninsula Standard Time 2006 Row: 1 Page: 1

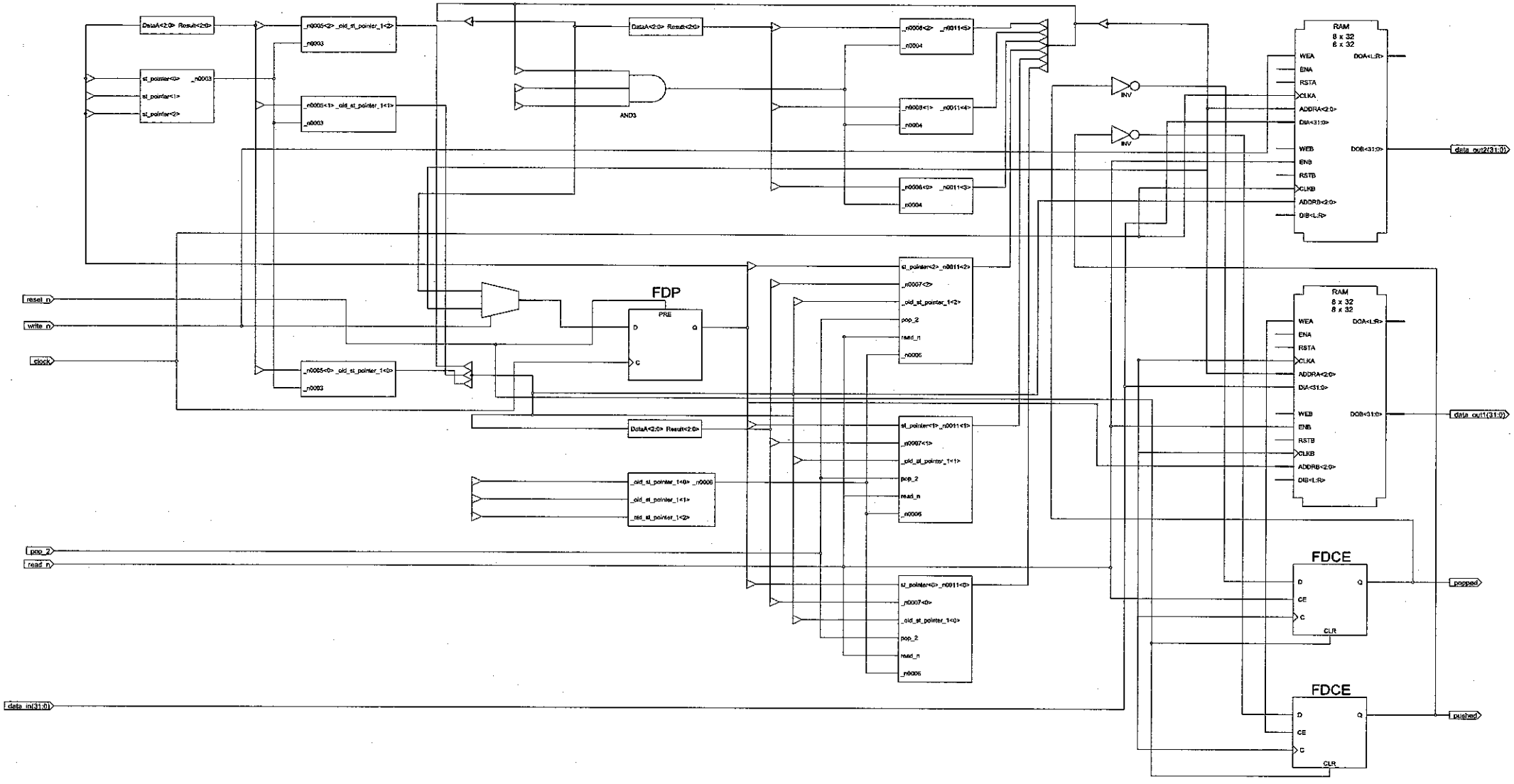


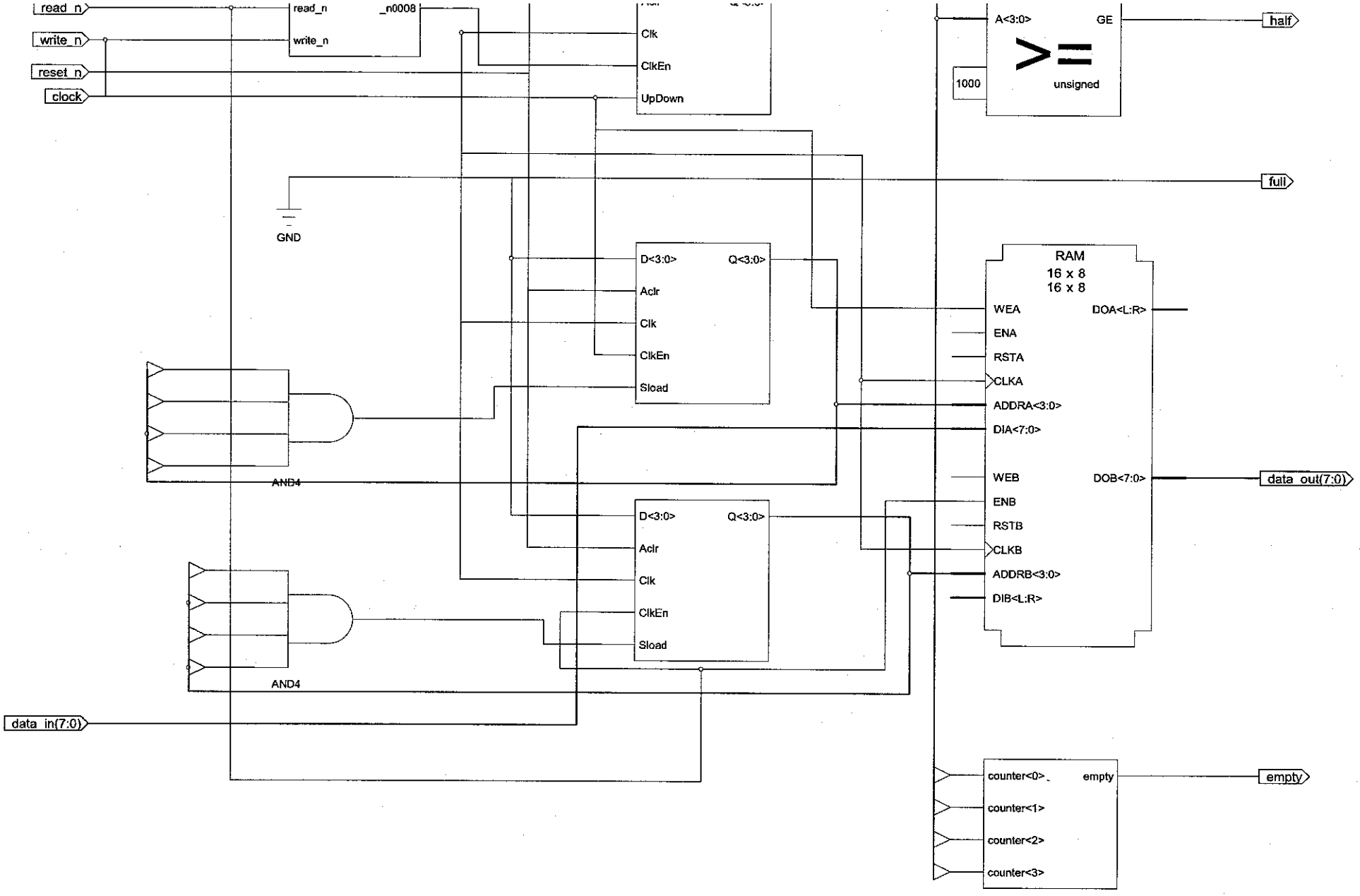
Entity:MJava_tst Architecture: Date: Wed May 10 12:40:59 Malay Peninsula Standard Time 2006 Row: 1 Page: 1



Entity:MJava_tst Architecture: Date: Wed May 10 12:40:59 Malay Peninsula Standard Time 2006 Row: 1 Page: 2

APPENDIX C: MJAVA STACKS SYNTHESIZED CIRCUIT


FDP
PRE
D
Q
C
AND3
INV
INV
RAM
8 x 32
8 x 32
WEA
ENA
RSTA
CLKA
ADDRA<2:0>
DIA<31:0>
DOA<L:R>
WEB
ENB
RSTB
CLKB
ADDRB<2:0>
DIB<L:R>
DOB<31:0>
data_out2(31:0)
data_out1(31:0)
FDCE
CE
CLR
popped
pushed
resel_n
write_n
clock
pop_2
read_n
data_in(31:0)


read_n
write_n
reset_n
clock
read_n
write_n
_n0008
Clk
ClkEn
UpDown
A<3:0>
GE
1000
unsigned
half
full
GND
D<3:0>
Q<3:0>
Aclr
Clk
ClkEn
Sload
RAM
16 x 8
16 x 8
WEA
DOA<L:R>
ENA
RSTA
CLKA
ADDRA<3:0>
DIA<7:0>
WEB
DOB<7:0>
ENB
RSTB
CLKB
ADDRB<3:0>
DIB<L:R>
data_out(7:0)
AND4
AND4
data_in(7:0)
counter<0>
counter<1>
counter<2>
counter<3>
empty

APPENDIX D1: JVM INSTRUCTIONS HEXADECIMAL VALUES

CHAPTER 10

Opcode Mnemonics by Opcode

0 (0x00) ..... *nop*
1 (0x01) ..... *aconst_null*
2 (0x02) ..... *iconst_m1*
3 (0x03) ..... *iconst_0*
4 (0x04) ..... *iconst_1*
5 (0x05) ..... *iconst_2*
6 (0x06) ..... *iconst_3*
7 (0x07) ..... *iconst_4*
8 (0x08) ..... *iconst_5*
9 (0x09) ..... *lconst_0*
10 (0x0a) ..... *lconst_1*
11 (0x0b) ..... *fconst_0*
12 (0x0c) ..... *fconst_1*
13 (0x0d) ..... *fconst_2*
14 (0x0e) ..... *dconst_0*
15 (0x0f) ..... *dconst_1*
16 (0x10) ..... *bipush*
17 (0x11) ..... *sipush*
18 (0x12) ..... *ldc*
19 (0x13) ..... *ldc_w*
20 (0x14) ..... *ldc2_w*
21 (0x15) ..... *iload*
22 (0x16) ..... *lload*
23 (0x17) ..... *fload*
24 (0x18) ..... *dload*
25 (0x19) ..... *aload*
26 (0x1a) ..... *iload_0*
27 (0x1b) ..... *iload_1*
28 (0x1c) ..... *iload_2*
29 (0x1d) ..... *iload_3*
30 (0x1e) ..... *lload_0*
31 (0x1f) ..... *lload_1*
32 (0x20) ..... *lload_2*
33 (0x21) ..... *lload_3*
34 (0x22) ..... *fload_0*
35 (0x23) ..... *fload_1*
36 (0x24) ..... *fload_2*
37 (0x25) ..... *fload_3*
38 (0x26) ..... *dload_0*
39 (0x27) ..... *dload_1*
40 (0x28) ..... *dload_2*
41 (0x29) ..... *dload_3*
42 (0x2a) ..... *aload_0*
43 (0x2b) ..... *aload_1*
44 (0x2c) ..... *aload_2*
45 (0x2d) ..... *aload_3*
46 (0x2e) ..... *iaload*
47 (0x2f) ..... *laload*
48 (0x30) ..... *faload*
49 (0x31) ..... *daload*
50 (0x32) ..... *aaload*
51 (0x33) ..... *baload*
52 (0x34) ..... *caload*
53 (0x35) ..... *saload*
54 (0x36) ..... *istore*
55 (0x37) ..... *lstore*

56 (0x38) ..... *fstore*
57 (0x39) ..... *dstore*
58 (0x3a) ..... *astore*
59 (0x3b) ..... *istore_0*
60 (0x3c) ..... *istore_1*
61 (0x3d) ..... *istore_2*
62 (0x3e) ..... *istore_3*
63 (0x3f) ..... *lstore_0*
64 (0x40) ..... *lstore_1*
65 (0x41) ..... *lstore_2*
66 (0x42) ..... *lstore_3*
67 (0x43) ..... *fstore_0*
68 (0x44) ..... *fstore_1*
69 (0x45) ..... *fstore_2*
70 (0x46) ..... *fstore_3*
71 (0x47) ..... *dstore_0*
72 (0x48) ..... *dstore_1*
73 (0x49) ..... *dstore_2*
74 (0x4a) ..... *dstore_3*
75 (0x4b) ..... *astore_0*
76 (0x4c) ..... *astore_1*
77 (0x4d) ..... *astore_2*
78 (0x4e) ..... *astore_3*
79 (0x4f) ..... *iastore*
80 (0x50) ..... *lastore*
81 (0x51) ..... *fastore*
82 (0x52) ..... *dastore*
83 (0x53) ..... *aastore*
84 (0x54) ..... *bastore*
85 (0x55) ..... *castore*
86 (0x56) ..... *sastore*
87 (0x57) ..... *pop*
88 (0x58) *pop2*
89 (0x59) ..... *dup*
90 (0x5a) ..... *dup_x1*
91 (0x5b) ..... *dup_x2*
92 (0x5c) *dup2*
93 (0x5d) ..... *dup2_x1*
94 (0x5e) ..... *dup2_x2*
95 (0x5f) ..... *swap*
96 (0x60) ..... *iadd*
97 (0x61) ..... *ladd*
98 (0x62) ..... *fadd*
99 (0x63) ..... *dadd*
100 (0x64) ..... *isub*
101 (0x65) ..... *lsub*
102 (0x66) ..... *fsub*
103 (0x67) ..... *dsub*
104 (0x68) ..... *imul*
105 (0x69) ..... *lmul*
106 (0x6a) ..... *fmul*
107 (0x6b) ..... *dmul*
108 (0x6c) ..... *idiv*
109 (0x6d) ..... *ldiv*
100 (0x6e) ..... *fdiv*
111 (0x6f) ..... *ddiv*
112 (0x70) ..... *irem*
113 (0x71) ..... *lrem*
114 (0x72) ..... *frem*
115 (0x73) ..... *drem*
116 (0x74) ..... *ineg*
117 (0x75) ..... *lneg*
118 (0x76) ..... *fneg*
119 (0x77) ..... *dneg*
120 (0x78) ..... *ishl*
121 (0x79) ..... *lshl*
122 (0x7a) ..... *ishr*
123 (0x7b) ..... *lshr*
124 (0x7c) ..... *iushr*
125 (0x7d) ..... *lushr*
126 (0x7e) ..... *iand*
127 (0x7f) ..... *land*
128 (0x80) ..... *ior*
129 (0x81) ..... *lor*
130 (0x82) ..... *ixor*
131 (0x83) ..... *lxor*
132 (0x84) ..... *iinc*
133 (0x85) *i2l*
134 (0x86) *i2f*
135 (0x87) *i2d*
136 (0x88) *l2i*
137 (0x89) *l2f*

138 (0x8a) *l2d*
139 (0x8b) *f2i*
140 (0x8c) *f2l*
141 (0x8d) *f2d*
142 (0x8e) *d2i*
143 (0x8f) *d2l*
144 (0x90) *d2f*
145 (0x91) *i2b*
146 (0x92) *i2c*
147 (0x93) *i2s*
148 (0x94) ..... *lcmp*
149 (0x95) ..... *fcmpl*
150 (0x96) ..... *fcmpg*
151 (0x97) ..... *dcmpl*
152 (0x98) ..... *dcmpg*
153 (0x99) ..... *ifeq*
154 (0x9a) ..... *ifne*
155 (0x9b) ..... *iflt*
156 (0x9c) ..... *ifge*
157 (0x9d) ..... *ifgt*
158 (0x9e) ..... *ifle*
159 (0x9f) ..... *if_icmpeq*
160 (0xa0) ..... *if_icmpne*
161 (0xa1) ..... *if_icmplt*
162 (0xa2) ..... *if_icmpge*
163 (0xa3) ..... *if_icmpgt*
164 (0xa4) ..... *if_icmple*
165 (0xa5) ..... *if_acmpeq*
166 (0xa6) ..... *if_acmpne*
167 (0xa7) ..... *goto*
168 (0xa8) ..... *jsr*
169 (0xa9) ..... *ret*
170 (0xaa) ..... *tableswitch*
171 (0xab) ..... *lookupswitch*
172 (0xac) ..... *ireturn*
173 (0xad) ..... *lreturn*
174 (0xae) ..... *freturn*
175 (0xaf) ..... *dreturn*
176 (0xb0) ..... *areturn*
177 (0xb1) ..... *return*
178 (0xb2) ..... *getstatic*
179 (0xb3) ..... *putstatic*
180 (0xb4) ..... *getfield*
181 (0xb5) ..... *putfield*
182 (0xb6) ..... *invokevirtual*
183 (0xb7) ..... *invokespecial*
184 (0xb8) ..... *invokestatic*
185 (0xb9) ..... *invokeinterface*
186 (0xba) ..... *xxxunusedxxx*
187 (0xbb) ..... *new*
188 (0xbc) ..... *newarray*
189 (0xbd) ..... *anewarray*
190 (0xbe) ..... *arraylength*
191 (0xbf) ..... *athrow*
192 (0xc0) ..... *checkcast*
193 (0xc1) ..... *instanceof*
194 (0xc2) ..... *monitorenter*
195 (0xc3) ..... *monitorexit*
196 (0xc4) ..... *wide*
197 (0xc5) ..... *multianewarray*
198 (0xc6) ..... *ifnull*
199 (0xc7) ..... *ifnonnull*
200 (0xc8) ..... *goto_w*
201 (0xc9) ..... *jsr_w*

_quick opcodes:

203 (0xcb) ..... *ldc_quick*
204 (0xcc) ..... *ldc_w_quick*
205 (0xcd) ..... *ldc2_w_quick*
206 (0xce) ..... *getfield_quick*
207 (0xcf) ..... *putfield_quick*
208 (0xd0) ..... *getfield2_quick*
209 (0xd1) ..... *putfield2_quick*
210 (0xd2) ..... *getstatic_quick*
211 (0xd3) ..... *putstatic_quick*
212 (0xd4) ..... *getstatic2_quick*
213 (0xd5) ..... *putstatic2_quick*
214 (0xd6) ..... *invokevirtual_quick*
215 (0xd7) .... *invokenonvirtual_quick*
216 (0xd8) ..... *invokesuper_quick*
217 (0xd9) ..... *invokestatic_quick*
218 (0xda) ..... *invokeinterface_quick*

219 (0xdb). *invokevirtualobject_quick*
221 (0xdd)..... *new_quick*
222 (0xde)..... *anewarray_quick*
223 (0xdf)..... *multianewarray_quick*
224 (0xe0)..... *checkcast_quick*
225 (0xe1)..... *instanceof_quick*
226 (0xe2)..... *invokevirtual_quick_w*
227 (0xe3)..... *getfield_quick_w*
228 (0xe4)..... *putfield_quick_w*

Reserved opcodes:
202 (0xca)..... *breakpoint*
254 (0xfe)..... *impdep1*
255 (0xff)..... *impdep2*

APPENDIX D2: JVM INSTRUCTIONS AND OPERANDS DESCRIPTION

mnemonic ***mnemonic***

Operation Short description of the instruction

Format

mnemonic
operand1
operand2
...

Operation

Forms mnemonic = opcode

Stack ..., value1, value2 ⇒
..., value3

Description A longer description detailing constraints on operand stack contents or constant pool entries, the operation performed, the type of the results, etc.

Linking Exceptions If any linking exceptions may be thrown by the execution of this instruction they are set off one to a line, in the order in which they must be thrown.

Runtime Exceptions If any runtime exceptions can be thrown by the execution of an instruction they are set off one to a line, in the order in which they must be thrown.

Other than the linking and runtime exceptions, if any, listed for an instruction, that instruction must not throw any runtime exceptions except for instances of `VirtualMachineError` or its subclasses.

Notes Comments not strictly part of the specification of an instruction are set aside as notes at the end of the description.

Figure 6.1 An example instruction page

Each cell in the instruction format diagram represents a single 8-bit byte. The instruction's mnemonic is its name. Its opcode is its numeric representation and is

given in both decimal and hexadecimal forms. Only the numeric representation is actually present in the Java Virtual Machine code in a `class` file.

Keep in mind that there are "operands" generated at compile time and embedded within Java Virtual Machine instructions, as well as "operands" calculated at run time and supplied on the operand stack. Although they are supplied from several different areas, all these operands represent the same thing: values to be operated upon by the Java Virtual Machine instruction being executed. By implicitly taking many of its operands from its operand stack, rather than representing them explicitly in its compiled code as additional operand bytes, register numbers, etc., the Java Virtual Machine's code stays compact.

Some instructions are presented as members of a family of related instructions sharing a single description, format, and operand stack diagram. As such, a family of instructions includes several opcodes and opcode mnemonics; only the family mnemonic appears in the instruction format diagram, and a separate forms line lists all member mnemonics and opcodes. For example, the forms line for the *lconst_<l>* family of instructions, giving mnemonic and opcode information for the two instructions in that family (*lconst_0* and *lconst_1*), is

Forms *lconst_0* = 9 (0x9),
lconst_1 = 10 (0xa)

In the description of the Java Virtual Machine instructions, the effect of an instruction's execution on the operand stack (§3.6.2) of the current frame (§3.6) is represented textually, with the stack growing from left to right and each word (§3.4) represented separately. Thus,

Stack ..., *value1*, *value2* ⇒
..., *result*

shows an operation that begins by having a one-word *value2* on top of the operand stack with a one-word *value1* just beneath it. As a result of the execution of the instruction, *value1* and *value2* are popped from the operand stack and replaced by a one-word *result*, which has been calculated by the instruction. The remainder of the operand stack, represented by an ellipsis (...), is unaffected by the instruction's execution.

The types `long` and `double` take two words on the operand stack. In the operand stack representation, each word is represented separately using a dot notation:

Stack ..., *value1 word1, value1 word2, value2 word1, value2 word2* ⇒
..., *result word1, result word2*

The Java Virtual Machine specification does not mandate how the two words are used to represent the 64-bit `long` or `double` value; it only requires that a particular implementation be internally consistent.

bipush *bipush*

Operation Push `byte`

Format

bipush
byte

Forms *bipush* = 16 (0x10)

Stack ... ⇒
..., *value*

Description The immediate *byte* is sign-extended to an `int`, and the resulting *value* is pushed onto the operand stack.

dup *dup*

Operation Duplicate top operand stack word

Format

dup

Forms *dup* = 89 (0x59)

Stack ..., *word* ⇒
..., *word*, *word*

Description The top word on the operand stack is duplicated and pushed onto the operand stack.

The *dup* instruction must not be used unless *word* contains a 32-bit data type.

Notes Except for restrictions preserving the integrity of 64-bit data types, the *dup* instruction operates on an untyped word, ignoring the type of the datum it contains.

dup2 *dup2*

Operation Duplicate top two operand stack words

Format

dup2

Forms *dup2* = 92 (0x5c)

Stack ..., *word2*, *word1* ⇒
..., *word2*, *word1*, *word2*, *word1*

Description The top two words on the operand stack are duplicated and pushed onto the operand stack, in the original order.

The *dup2* instruction must not be used unless each of *word1* and *word2* is a word that contains a 32-bit data type or both together are the two words of a single 64-bit datum.

Notes Except for restrictions preserving the integrity of 64-bit data types, the *dup2* instruction operates on untyped words, ignoring the types of the data they contain.

goto *goto*

Operation Branch always

Format

goto
branchbyte1
branchbyte2

Forms *goto* = 167 (0xa7)

Stack No change

Description The unsigned bytes *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit *branchoffset*, where *branchoffset* is (*branchbyte1* << 8) | *branchbyte2*. Execution proceeds at that offset from the address of the opcode of this *goto* instruction. The target address must be that of an opcode of an instruction within the method that contains this *goto* instruction.

iadd *iadd*

Operation Add `int`

Format

iadd

Forms *iadd* = 96 (0x60)

Stack ..., *value1*, *value2* ⇒
..., *result*

Description Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

If an *iadd* overflows, then the result is the low-order bits of the true mathematical result in a sufficiently wide two's-complement format. If overflow occurs, then the sign of the result will not be the same as the sign of the mathematical sum of the two values.

iand *iand*

Operation Boolean AND `int`

Format

iand

Forms *iand* = 126 (0x7e)

Stack ..., *value1*, *value2* ⇒
..., *result*

Description Both *value1* and *value2* must be of type `int`. They are popped from the operand stack. An `int` *result* is calculated by taking the bitwise AND (conjunction) of *value1* and *value2*. The *result* is pushed onto the operand stack.

iconst_<i> *iconst_<i>*

Operation Push `int` constant

Format

iconst_<i>

Forms

iconst_m1 = 2 (0x2)
iconst_0 = 3 (0x3)
iconst_1 = 4 (0x4)
iconst_2 = 5 (0x5)
iconst_3 = 6 (0x6)
iconst_4 = 7 (0x7)
iconst_5 = 8 (0x8)

Stack

... ⇒
..., <i>

Description Push the `int` constant <i> (*−1*, *0*, *1*, *2*, *3*, *4* or *5*) onto the operand stack.

Notes Each of this family of instructions is equivalent to *bipush* <i> for the respective value of <i>, except that the operand <i> is implicit.

if_icmp<cond> *if_icmp<cond>*

Operation Branch if `int` comparison succeeds

Format

if_icmp<cond>
branchbyte1
branchbyte2

Forms

if_icmpeq = 159 (0x9f)
if_icmpne = 160 (0xa0)
if_icmplt = 161 (0xa1)
if_icmpge = 162 (0xa2)
if_icmpgt = 163 (0xa3)
if_icmple = 164 (0xa4)

Stack

..., *value1*, *value2* ⇒
...

Description Both *value1* and *value2* must be of type `int`. They are both popped from the operand stack and compared. All comparisons are signed. The results of the comparison are as follows:

- *eq* succeeds if and only if *value1* = *value2*
- *ne* succeeds if and only if *value1* ≠ *value2*
- *lt* succeeds if and only if *value1* < *value2*
- *le* succeeds if and only if *value1* ≤ *value2*
- *gt* succeeds if and only if *value1* > *value2*
- *ge* succeeds if and only if *value1* ≥ *value2*

if_icmp<cond> (cont.) *if_icmp<cond> (cont.)*

If the comparison succeeds, the unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is calculated to be (*branchbyte1* << 8) | *branchbyte2*. Execution then proceeds at that offset from the address of the opcode of this *if_icmp<cond>* instruction. The target address must be that of an opcode of an instruction within the method that contains this *if_icmp<cond>* instruction.

Otherwise, execution proceeds at the address of the instruction following this *if_icmp<cond>* instruction.

iinc *iinc*

Operation Increment local variable by constant

Format

iinc
index
const

Forms *iinc* = 132 (0x84)

Stack No change

Description The *index* is an unsigned byte that must be a valid index into the local variables of the current frame (§3.6). The *const* is a immediate signed byte. The local variable at *index* must contain an `int`. The value *const* is first sign-extended to an `int`, then the local variable at *index* is incremented by that amount.

Notes The *iinc* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index and increment it by a two-byte immediate value.

iload *iload*

Operation Load `int` from local variable

Format

iload
index

Forms *iload* = 21 (0x15)

Stack ... ⇒
..., *value*

Description The *index* is an unsigned byte that must be a valid index into the local variables of the current frame (§3.6). The local variable at *index* must contain an `int`. The *value* of the local variable at *index* is pushed onto the operand stack.

Notes The *iload* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

iload_<n> *iload_<n>*

Operation Load `int` from local variable

Format

iload_<n>

Forms *iload_0* = 26 (0x1a)
iload_1 = 27 (0x1b)
iload_2 = 28 (0x1c)
iload_3 = 29 (0x1d)

Stack ... ⇒
..., *value*

Description The *<n>* must be a valid index into the local variables of the current frame (§3.6). The local variable at *<n>* must contain an `int`. The *value* of the local variable at *<n>* is pushed onto the operand stack.

Notes Each of the *iload_<n>* instructions is the same as *iload* with an *index* of *<n>*, except that the operand *<n>* is implicit.

ineg *ineg*

Operation Negate `int`

Format

ineg

Forms *ineg* = 116 (0x74)

Stack ..., *value* ⇒
..., *result*

Description The *value* must be of type `int`. It is popped from the operand stack. The `int` *result* is the arithmetic negation of *value*, *–value*. The *result* is pushed onto the operand stack.

For `int` values, negation is the same as subtraction from zero. Because the Java Virtual Machine uses two's-complement representation for integers and the range of two's-complement values is not symmetric, the negation of the maximum negative `int` results in that same maximum negative number. Despite the fact that overflow has occurred, no exception is thrown.

For all `int` values `x`, `-x` equals `(~x) + 1`.

ior *ior*

Operation Boolean OR `int`

Format

ior

Forms *ior* = 128 (0x80)

Stack ..., *value1*, *value2* ⇒
..., *result*

Description Both *value1* and *value2* must both be of type `int`. They are popped from the operand stack. An `int` *result* is calculated by taking the bitwise inclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

ishl *ishl*

Operation Shift left `int`

Format

ishl

Forms *ishl* = 120 (0x78)

Stack ..., *value1*, *value2* ⇒
..., *result*

Description Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by shifting *value1* left by *s* bit positions, where *s* is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

Notes This is equivalent (even if overflow occurs) to multiplication by 2 to the power *s*. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

ishr *ishr*

Operation Arithmetic shift right `int`

Format

ishr

Forms *ishr* = 122 (0x7a)

Stack ..., *value1*, *value2* ⇒
..., *result*

Description Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. An `int` *result* is calculated by shifting *value1* right by *s* bit positions, with sign extension, where *s* is the value of the low five bits of *value2*. The *result* is pushed onto the operand stack.

Notes The resulting value is $\lfloor (value1)/2^s \rfloor$, where *s* is *value2* & 0x1f. For nonnegative *value1*, this is equivalent to truncating `int` division by 2 to the power `s`. The shift distance actually used is always in the range 0 to 31, inclusive, as if *value2* were subjected to a bitwise logical AND with the mask value 0x1f.

istore *istore*

Operation Store `int` into local variable

Format

istore
index

Forms *istore* = 54 (0x36)

Stack ..., *value* ⇒
...

Description The *index* is an unsigned byte that must be a valid index into the local variables of the current frame (§3.6). The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack, and the value of the local variable at *index* is set to *value*.

Notes The *istore* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

istore_<n> *istore_<n>*

Operation Store `int` into local variable

Format | *istore_<n>* |

Forms *istore_0* = 59 (0x3b)
istore_1 = 60 (0x3c)
istore_2 = 61 (0x3d)
istore_3 = 62 (0x3e)

Stack ..., *value* ⇒
...

Description The <*n*> must be a valid index into the local variables of the current frame (§3.6). The *value* on the top of the operand stack must be of type `int`. It is popped from the operand stack, and the value of the local variable at <*n*> is set to *value*.

Notes Each of the *istore_<n>* instructions is the same as *istore* with an *index* of <*n*>, except that the operand <*n*> is implicit.

isub *isub*

Operation Subtract `int`

Format

isub

Forms *isub* = 100 (0x64)

Stack ..., *value1*, *value2* ⇒
..., *result*

Description Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* – *value2*. The *result* is pushed onto the operand stack.

For `int` subtraction, `a – b` produces the same result as `a + (–b)`. For `int` values, subtraction from zero is the same as negation.

Despite the fact that overflow or underflow may occur, in which case the *result* may have a different sign than the true mathematical result, execution of an *isub* instruction never throws a runtime exception.

ixor *ixor*

Operation Boolean XOR `int`

Format

ixor

Forms *ixor* = 130 (0x82)

Stack ..., *value1*, *value2* ⇒
..., *result*

Description Both *value1* and *value2* must both be of type `int`. They are popped from the operand stack. An `int` *result* is calculated by taking the bitwise exclusive OR of *value1* and *value2*. The *result* is pushed onto the operand stack.

jsr *jsr*

Operation Jump subroutine

Format

jsr
branchbyte1
branchbyte2

Forms *jsr* = 168 (0xa8)

Stack ... ⇒
..., *address*

Description The *address* of the opcode of the instruction immediately following this *jsr* instruction is pushed onto the operand stack as a value of type `returnAddress`. The unsigned *branchbyte1* and *branchbyte2* are used to construct a signed 16-bit offset, where the offset is (*branchbyte1* << 8) | *branchbyte2*. Execution proceeds at that offset from the address of this *jsr* instruction. The target address must be that of an opcode of an instruction within the method that contains this *jsr* instruction.

Notes The *jsr* instruction is used with the *ret* instruction in the implementation of the `finally` clauses of the Java language (see Section 7.13, "Compiling `finally`"). Note that *jsr* pushes the address onto the stack and *ret* gets it out of a local variable. This asymmetry is intentional.

nop *nop*

Operation Do nothing

Format

nop

Forms *nop* = 0 (0x0)

Stack No change

Description Do nothing.

pop *pop*

Operation Pop top operand stack word

Format

pop

Forms *pop* = 87 (0x57)

Stack ..., *word* ⇒
...

Description The top word is popped from the operand stack.

The *pop* instruction must not be used unless *word* is a word that contains a 32-bit data type.

Notes Except for restrictions preserving the integrity of 64-bit data types, the *pop* instruction operates on an untyped word, ignoring the type of the datum it contains.

pop2 *pop2*

Operation Pop top two operand stack words

Format

pop2

Forms *pop2* = 88 (0x58)

Stack ..., *word2*, *word1* ⇒
...

Description The top two words are popped from the operand stack.

The *pop2* instruction must not be used unless each of word *word1* and *word2* is a word that contains a 32-bit data types or together are the two words of a single 64-bit datum.

Notes Except for restrictions preserving the integrity of 64-bit data types, the *pop2* instruction operates on raw words, ignoring the types of the data they contain.

ret *ret*

Operation Return from subroutine

Format

ret
index

Forms *ret* = 169 (0xa9)

Stack No change

Description The *index* is an unsigned byte between 0 and 255, inclusive. The local variable at *index* in the current frame (§3.6) must contain a value of type `returnAddress`. The contents of the local variable are written into the Java Virtual Machine's `pc` register, and execution continues there.

Notes The *ret* instruction is used with *jsr* or *jsr_w* instructions in the implementation of the `finally` keyword of the Java language (see Section 7.13, "Compiling `finally`"). Note that *jsr* pushes the address onto the stack and *ret* gets it out of a local variable. This asymmetry is intentional.

The *ret* instruction should not be confused with the *return* instruction. A *return* instruction returns control from a Java method to its invoker, without passing any value back to the invoker.

The *ret* opcode can be used in conjunction with the *wide* instruction to access a local variable using a two-byte unsigned index.

sipush *sipush*

Operation Push `short`

Format

sipush
byte1
byte2

Forms *sipush* = 17 (0x11)

Stack
... ⇒
..., *value*

Description The immediate unsigned *byte1* and *byte2* values are assembled into an intermediate `short` where the value of the short is (*byte1* << 8) | *byte2*. The intermediate value is then sign-extended to an `int`, and the resulting *value* is pushed onto the operand stack.

swap *swap*

Operation Swap top two operand stack words

Format

swap

Forms *swap* = 95 (0x5f)

Stack ..., *word2*, *word1* ⇒
..., *word1*, *word2*

Description The top two words on the operand stack are swapped.

The *swap* instruction must not be used unless each of *word2* and *word1* is a word that contains a 32-bit data type.

Notes Except for restrictions preserving the integrity of 64-bit data types, the *swap* instruction operates on untyped words, ignoring the types of the data they contain.