PROJECT REPORT

ON

"DESIGN OF A MICROPROCESSOR"

SUBMITTED FOR THE PARTIAL FULFILMENT OF THE DEGREE

OF

BACHELOR OF ENGINEERING IN

ELECTRONICS AND COMPUTER ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING
M.B.M. ENGINEERING COLLEGE, FACULTY OF ENGINEERING AND ARCHITECTURE, JAI NARAIN VYAS UNIVERSITY,
JODHPUR (RAJASTHAN)
SESSION 2016-17

SUBMITTED BY:

Tarun Malviya, 13/19514, 14ECC76044

Kanishk Singhvi 13/19494, 14ECC76019

Anushika Makhija, 13/19068, 14ECC76006

Latika Raina, 13/19027, 14ECC76023

GUIDED BY:

Miss Renu Bhardwaj

DECLARATION

We, **Tarun Malviya, Kanishk Singhvi, Anushika Makhija** and **Latika Raina** hereby declare that this project titled **"Design of a Microprocessor"** is a record of original work done by us under the supervision and guidance of **Miss Renu Bhardwaj**. We further certify that this project work has not formed the basis for the award of the Degree/ Diploma/ Associateship/ Fellowship or similar work to any candidate of any university and no part of this report is reproduced as it is from any other source without seeking permission.

Tarun Malviya
(14ECC76044)

Kanishk Singhvi
(14ECC76019)

Anushika Makhija
(14ECC76006)

Latika Raina
(14ECC76023)

ACKNOWLEDGMENT

The satiation and euphoria that accompany the successful completion of project would be incomplete without the mention of the people who made it possible.

We express our deep sense of gratitude to **Dr. V. S. Chauhan**, Head of Department, Electronics and Communication Engineering Department, M.B.M. Engineering College, Faculty of Engineering and Architecture, Jai Narain Vyas University, Jodhpur, for encouraging us to take this project.

We express deep and sincere gratitude to **Miss Renu Bhardwaj**, under whose guidance, encouragement, suggestion and very constructive criticism have contributed immensely to the evolution of our ideas on the project.

We acknowledge with thanks the kind of patronage, loving inspiration and timely guidance, which we have received from **Mr. Alok Singh Gehlot**, assistant professor, Computer Science Department, and **Dr. Rajesh Purohit**, Head of Department, Computer Science Department, M.B.M. Engineering College, Faculty of Engineering and Architecture, Jai Narain Vyas University, Jodhpur. We are highly indebted for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.

Our thanks and appreciations also go to all the colleagues and people, who have willingly helped us out with their abilities in developing the project.

Tarun Malviya (14ECC76044)
Kanishk Singhvi (14ECC76019)
Anushika Makhija (14ECC76006)
Latika Raina (14ECC76023)

CERTIFICATE

This is to certify that report entitled **"*Design of a Microprocessor*"** submitted to Department of Electronics and Communication Engineering, MBM Engineering College, Jai Narain Vyas University in partial fulfilment of the requirements for the award of the degree of Bachelor of Engineering in Electronics and Computer is the bonafied record of the work done by **Tarun Malviya, Kanishk Singhvi, Anushika Makhija and Latika Raina** under my supervision and guidance.

Dr. V. S. Chauhan
Head Of Department
Electronics and Communication Engineering Department
M.B.M. Engineering College, JNVU, Jodhpur

ABSTRACT

We present the design and evaluation of a 32-bit microprocessor using Verilog HDL. More precisely, we will design a subset machine of Java Virtual Machine (JVM). This subset Instruction Set Architecture (ISA) contains only integer instructions, so we named it IJVM. We will start by describing the microarchitecture on the top of which we will implement IJVM. IJVM has some relatively complex instructions. Many such instructions have often be implemented through microprogramming.

Our microarchitecture will contain a microprogram (in ROM), whose job is to fetch, decode, and execute IJVM instructions.

The microprogram has a set of variables, called the state of the computer, which can be accessed by all the functions. Each function changes at least some of the variables making up the state. For example, the Program Counter (PC) is part of the state. It indicates the memory location containing the next function (i.e., ISA instruction) to be executed. During the execution of each instruction, the PC is advanced to point to the next instruction to be executed. IJVM instructions are short and sweet. Each instruction has a few fields, usually one or two, each of which has some specific purpose. The first field of every instruction is the opcode (short for operation code), which identifies the instruction, telling whether it is an ADD or a BRANCH, or something else. Many instructions have an additional field, which specifies the operand. For example, instructions that access a local variable need a field to tell which variable. This model of execution, sometimes called the fetch-execute cycle, is useful in the abstract and may also be the basis for implementation for ISAs like IJVM that have complex instructions. Below we will describe how it works, what the microarchitecture looks like, and how it is controlled by the microinstructions, each of which controls the data path for one cycle. Together, the list of microinstructions forms the microprogram, which we will present and discuss in detail.

In the end we will discuss over the future modifications of this processor and conclude the report.

CONTENTS

1. INTRODUCTION

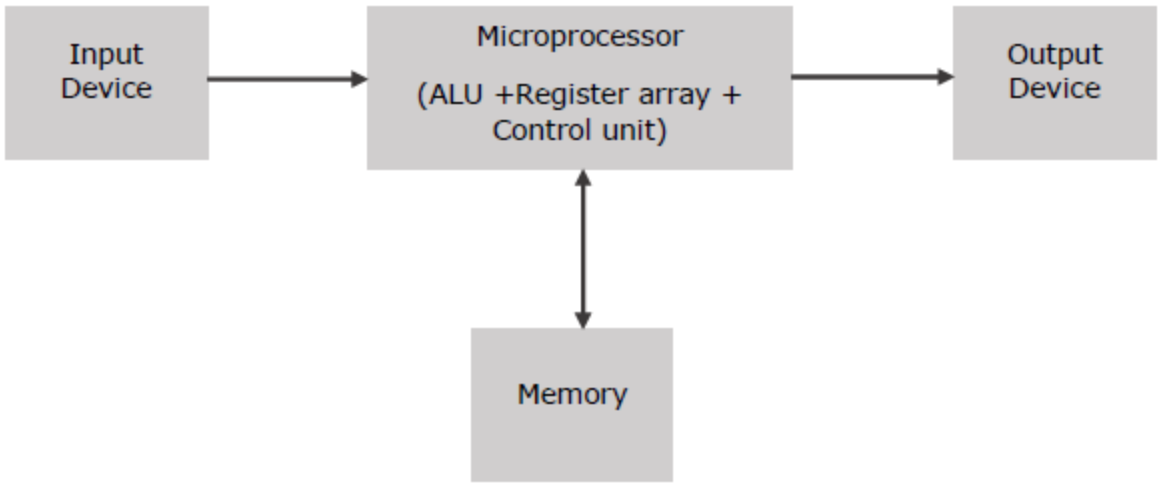
A microprocessor is a computer processor which incorporates the functions of a computer's central processing unit (CPU) on a single integrated circuit (IC), or at most a few integrated circuits. The microprocessor is a multipurpose, clock driven, register based, digital-integrated circuit which accepts binary data as input, processes it according to instructions stored in its memory, and provides results as output. Microprocessors contain both combinational logic and sequential digital logic. Microprocessors operate on numbers and symbols represented in the binary numeral system.

The integration of a whole CPU onto a single chip or on a few chips greatly reduced the cost of processing power, increasing efficiency. Integrated circuit processors are produced in large numbers by highly automated processes resulting in a low per unit cost. Single-chip processors increase reliability as there are many fewer electrical connections to fail. As microprocessor designs get better, the cost of manufacturing a chip (with smaller components built on a semiconductor chip the same size) generally stays the same.

Before microprocessors, small computers had been built using racks of circuit boards with many medium- and small-scale integrated circuits. Microprocessors combined this into one or a few large-scale ICs. Continued increases in microprocessor capacity have since rendered other forms of computers almost completely obsolete, with one or more microprocessors used in everything from the smallest embedded systems and handheld devices to the largest mainframes and supercomputers.

1.1 STRUCTURE

Microprocessor is a controlling unit of a micro-computer, fabricated on a small chip capable of performing ALU (Arithmetic Logical Unit) operations and communicating with the other devices connected to it. Microprocessor consists of an ALU, register array, and a control unit. ALU performs arithmetical and logical operations on the data received from the memory or an input device. Register array consists of registers identified by letters like B, C, D, E, H, L and accumulator. The control unit controls the flow of data and instructions within the computer.



Block Diagram of a Basic Microcomputer

The internal arrangement of a microprocessor varies depending on the age of the design and the intended purposes of the microprocessor. The complexity of an integrated circuit (IC) is bounded by physical limitations of the number of transistors that can be put onto one chip, the number of package terminations that can connect the processor to other parts of the system, the number of interconnections it is possible to make on the chip, and the heat that the chip can dissipate. Advancing technology makes more complex and powerful chips feasible to manufacture.

A minimal hypothetical microprocessor might only include an arithmetic logic unit (ALU) and a control logic section. The ALU performs operations such as addition, subtraction, and operations such as AND or OR. Each operation of the ALU sets one or more flags in a status register, which indicate the results of the last operation (zero value, negative number, overflow, or others). The control logic retrieves instruction codes from memory and initiates the sequence of operations required for the ALU to carry out the instruction. A single operation code might affect many individual data paths, registers, and other elements of the processor.

As integrated circuit technology advanced, it was feasible to manufacture more and more complex processors on a single chip. The size of data objects became larger; allowing more transistors on a chip allowed word sizes to increase from 4- and 8-bit words up to today's 64-bit words. Additional features were added to the processor architecture; more on-chip registers sped up programs, and complex instructions could be used to make more compact programs. Floating-point arithmetic, for example, was often not available on 8-bit microprocessors, but had to be carried out in software. Integration of the floating point unit first as a separate integrated circuit and then as part of the same microprocessor chip, sped up floating point calculations.

Occasionally, physical limitations of integrated circuits made such practices as a bit slice approach necessary. Instead of processing all of a long word on one integrated circuit, multiple circuits in parallel processed subsets of each data word. While this required extra logic to handle, for example, carry and overflow within each slice, the result was a system that could handle, for example, 32-bit words using integrated circuits with a capacity for only four bits each.

With the ability to put large numbers of transistors on one chip, it becomes feasible to integrate memory on the same die as the processor. This CPU cache has the advantage of faster access than off-chip memory, and increases the processing speed of the system for many applications. Processor clock frequency has increased more rapidly than external memory speed, except in the recent past, so cache memory is necessary if the processor is not delayed by slower external memory.

1.2 WORKING

The microprocessor follows a sequence: Fetch, Decode, and then Execute. Initially, the instructions are stored in the memory in a sequential order. The microprocessor fetches those instructions from the memory, then decodes it and executes those instructions till STOP instruction is reached. Later, it sends the result in binary to the output port. Between these processes, the register stores the temporarily data and ALU performs the computing functions.

1.3 FEATURES

- Cost-effective −The microprocessor chips are available at low prices and results its low cost.
- Size −The microprocessor is of small size chip, hence is portable.

- Low Power Consumption − Microprocessors are manufactured by using metal oxide semiconductor technology, which has low power consumption.
- Versatility −The microprocessors are versatile as we can use the same chip in a number of applications by configuring the software program.
- Reliability −The failure rate of an IC in microprocessors is very low, hence it is reliable.

1.4 APPLICATIONS OF A MICROPROCESSOR

The microprocessor, also known as the Central Processing Unit (CPU), is the brain of all computers and many household and electronic devices. Multiple microprocessors, working together, are the "hearts" of data centres, super-computers, communications products, and other digital devices. The first microprocessor was the Intel 4004, introduced in 1971. The 4004 was not very powerful; it was primarily used to perform simple mathematical operations in a calculator called “Busicom.” Just like microwaves or telephones, devices with microprocessors have become so integrated into our daily lives, that we cannot imagine a life without them. Thousands of items that were traditionally not computer-related include microprocessors. These include large and small household appliances, cars (and their accessory equipment units), car keys, tools and test instruments, toys, light switches/dimmers and electrical circuit breakers, smoke alarms, battery packs, and hi-fi audio/visual components (from DVD players to phonograph turntables). Such products as cellular telephones, DVD video system and HDTV broadcast systems fundamentally require consumer devices with powerful, low-cost, microprocessors. Increasingly stringent pollution control standards effectively require automobile manufacturers to use microprocessor engine management systems, to allow optimal control of emissions over widely varying operating conditions of an automobile. Non-programmable controls would require complex, bulky, or costly implementation to achieve the results possible with a microprocessor.

A microprocessor control program (embedded software) can be easily tailored to different needs of a product line, allowing upgrades in performance with minimal redesign of the product. Different features can be implemented in different models of a product line at negligible production cost.

Microprocessor control of a system can provide control strategies that would be impractical to implement using electromechanical controls or purpose-built electronic controls. For example, an engine control system in an automobile can adjust ignition timing based on engine speed, load on the engine, ambient temperature, and any observed tendency for knocking—allowing an automobile to operate on a range of fuel grades.

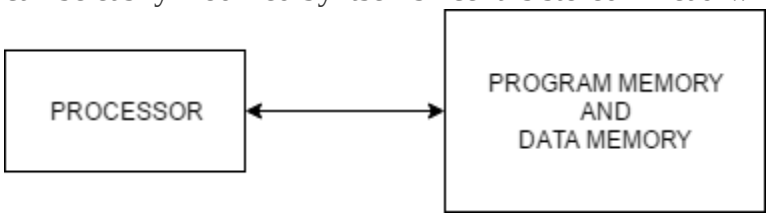
Nowadays, modern microprocessors can perform extremely sophisticated operations in areas such as meteorology, aviation, nuclear physics and engineering, and take up much less space as well as delivering superior performance. Over the past 40 years, microprocessors have become faster and more powerful, yet increasingly smaller and more affordable. The manufacturing of a CPU is a highly complex and demanding process involving multiple hundreds of steps in “cleanrooms.”

2. COMPUTER ARCHITECTURES

There are basically two types of digital computer architectures. The first one is called Von Neumann architecture and later Harvard architecture was adopted for designing digital computers.

2.1 Von Neumann Architecture:

The **von Neumann Architecture** is named after the mathematician and early computer scientist John von Neumann. Von Neumann machines have shared signals and memory for code and data. Thus, the program can be easily modified by itself since it is stored in read-write memory.



- Processor needs two clock cycles to complete an instruction. Pipelining the instructions is not possible with this architecture.
- In the first clock cycle the processor gets the instruction from memory and decodes it. In the next clock cycle the required data is taken from memory. For each instruction this cycle repeats and hence needs two cycles to complete an instruction.
- This is a relatively older architecture and was replaced by Harvard architecture.

2.2 Harvard Architecture:

The name **Harvard Architecture** comes from the Harvard Mark I relay-based computer. The most obvious characteristic of the Harvard Architecture has separate data and instruction busses, allowing transfers to be performed simultaneously on both busses. A **von Neumann architecture** has only one bus which is used for both data transfers and instruction fetches, and therefore data transfers and instruction fetches must be scheduled - they cannot be performed at the same time.. It is possible to access program memory and data memory simultaneously. Typically, code (or program) memory is read-only and data memory is read-write. Therefore, it is impossible for program contents to be modified by the program itself.

- Processor can complete an instruction in one cycle if appropriate pipelining strategies are implemented.
- In the first stage of pipeline the instruction to be executed can be taken from program memory. In the second stage of pipeline data is taken from the data memory using the decoded instruction or address.
- Most of the modern computing architectures are based on Harvard architecture. But the number of stages in the pipeline varies from system to system.

2.3 RISC ARCHITECTURE

The microcontroller architecture that utilizes small and highly optimized set of instructions is termed as the Reduced Instruction Set Computer or simply called as RISC. It is also called as LOAD/STORE architecture. In the late 1970s and early 1980s, RISC projects were primarily developed from Stanford, UC-Berkley and IBM. The John Coke of IBM research team developed RISC by reducing the number of instructions required for processing computations faster than the CISC. The RISC architecture is faster and the chips required for the manufacture of RISC architecture is also less expensive compared to the CISC architecture.

Typical Features of RISC Architecture

- Pipelining technique of RISC, executes multiple parts or stages of instructions simultaneously such that every instruction on the CPU is optimized. Hence, the RISC processors have Clock per Instruction of one cycle, and this is called as One Cycle Execution.
- It optimizes the usage of register with more number of registers in the RISC and more number of interactions within the memory can be prevented.
- Simple addressing modes, even complex addressing can be done by using arithmetic AND/ OR logical operations.
- It simplifies the compiler design by using identical general purpose registers which allows any register to be used in any context.
- For efficient usage of the registers and optimization of the pipelining uses, reduced instruction set is required.
- The number of bits used for the opcode is reduced.

Advantages of RISC processor architecture

- Because of the small set of instructions of RISC, high-level language compilers can produce more efficient code.
- RISC allows freedom of using the space on microprocessors because of its simplicity.
- Instead of using Stack, many RISC processors use the registers for passing arguments and holding the local variables.
- RISC functions uses only a few parameters, and the RISC processors cannot use the call instructions, and therefore, use a fixed length instructions which are easy to pipeline.
- The speed of the operation can be maximized and the execution time can be minimized.
- Very less number of instruction formats (less than four), a few number of instructions (around 150) and a few addressing modes (less than four) are needed.

Drawbacks of RISC processor architecture

- With the increase in length of the instructions, the complexity increases for the RISC processors to execute due to its character cycle per instruction.
- The performance of the RISC processors depends mostly on the compiler or programmer as the knowledge of the compiler plays a major role while converting the CISC code to a RISC code; hence, the quality of the generated code depends on the compiler.
- While rescheduling the CISC code to a RISC code, termed as a code expansion, will increase the size. And, the quality of this code expansion will again depend on the compiler, and also on the machine's instruction set.

- The first level cache of the RISC processors is also a disadvantage of the RISC, in which these processors have large memory caches on the chip itself. For feeding the instructions, they require very fast memory systems.

2.4 CISC ARCHITECTURE

The main intend of the CISC processor architecture is to complete task by using less number of assembly lines. For this purpose, the processor is built to execute a series of operations.

Features of CISC Architecture

- To simplify the computer architecture, CISC supports microprogramming.
- CISC have more number of predefined instructions which makes high level languages easy to design and implement.
- CISC consists of less number of registers and more number of addressing modes, generally 5 to 20.
- CISC processor takes varying cycle time for execution of instructions – multi-clock cycles.
- Because of the complex instruction set of the CISC, the pipelining technique is very difficult.
- CISC consists of more number of instructions, generally from 100 to 250.
- Special instructions are used very rarely.
- Operands in memory are manipulated by instructions.

Advantages of CISC architecture

- Each machine language instruction is grouped into a microcode instruction and executed accordingly, and then are stored inbuilt in the memory of the main processor, termed as microcode implementation.
- As the microcode memory is faster than the main memory, the microcode instruction set can be implemented without considerable speed reduction over hard wired implementation.
- Entire new instruction set can be handled by modifying the micro program design.
- CISC, the number of instructions required to implement a program can be reduced by building rich instruction sets and can also be made to use slow main memory more efficiently.
- Because of the superset of instructions that consists of all earlier instructions, this makes micro coding easy.

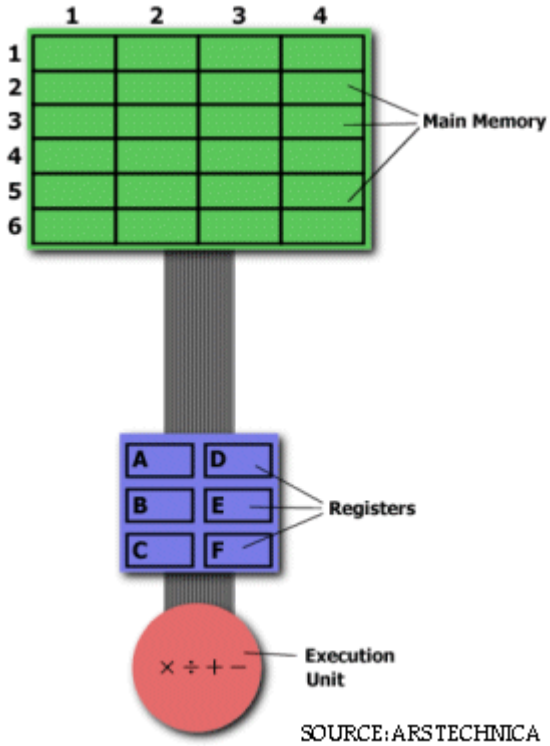
Drawbacks of CISC

- The amount of clock time taken by different instructions will be different – due to this – the performance of the machine slows down.
- The instruction set complexity and the chip hardware increases as every new version of the processor consists of a subset of earlier generations.
- Only 20% of the existing instructions are used in a typical programming event, even though there are many specialized instructions in existence which are not even used frequently.
- The conditional codes are set by the CISC instructions as a side effect of each instruction which takes time for this setting – and, as the subsequent instruction changes the condition code bits – so, the compiler has to examine the condition code bits before this happens.

The simplest way to examine the advantages and disadvantages of RISC architecture is by contrasting it with its predecessor: CISC (Complex Instruction Set Computers) architecture.

Multiplying Two Numbers in Memory
The diagram on the next page represents the storage scheme for a generic computer. The main memory is divided into locations numbered from (row) 1: (column) 1 to (row) 6: (column) 4. The execution unit is responsible for carrying out all computations. However, the execution unit can only operate on data that has been loaded into one of the six registers (A, B, C, D, E, or F). Let's say we want to find the product of two numbers - one stored in location 2:3 and another stored in location 5:2 - and then store the product back in the location 2:3.

The CISC Approach
The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations. For this particular task, a CISC processor would come prepared with a specific instruction (we'll call it "MULT"). When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register. Thus, the entire task of multiplying two numbers can be completed with one instruction:



MULT 2:3, 5:2

MULT is what is known as a "complex instruction." It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions. It closely resembles a command in a higher level language. For instance, if we let "a" represent the value of 2:3 and "b" represent the value of 5:2, then this command is identical to the C statement "a = a * b."

One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

The RISC Approach
RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MULT" command described above could be divided into three separate commands: "LOAD," which moves data from the memory bank to a register, "PROD," which finds the product of two operands located within the registers, and "STORE," which moves data from a register to the memory banks. In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A

At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form.

However, the RISC strategy also brings some very important advantages. Because each instruction requires only one clock cycle to execute, the entire program will execute in approximately the same amount of time as the multi-cycle "MULT" command. These RISC "reduced instructions" require less transistors of hardware space than the complex instructions, leaving more room for general purpose registers. Because all of the instructions execute in a uniform amount of time (i.e. one clock), pipelining is possible.

Separating the "LOAD" and "STORE" instructions actually reduces the amount of work that the computer must perform. After a CISC-style "MULT" command is executed, the processor automatically erases the registers. If one of the operands needs to be used for another computation, the processor must re-load the data from the memory bank into a register. In RISC, the operand will remain in the register until another value is loaded in its place.

The Performance Equation
The following equation is commonly used for expressing a computer's performance ability:

$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}$$

The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program.

RISC Roadblocks
Despite the advantages of RISC based processing, RISC chips took over a decade to gain a foothold in the commercial world. This was largely due to a lack of software support.

Although Apple's Power Macintosh line featured RISC-based chips and Windows NT was RISC compatible, Windows 3.1 and Windows 95 were designed with CISC processors in mind. Many companies were unwilling to take a chance with the emerging RISC technology. Without commercial interest, processor developers were unable to manufacture RISC chips in large enough volumes to make their price competitive.

Another major setback was the presence of Intel. Although their CISC chips were becoming increasingly unwieldy and difficult to develop, Intel had the resources to plow through development and produce powerful processors. Although RISC chips might surpass Intel's efforts in specific areas, the differences were not great enough to persuade buyers to change technologies.

The Overall RISC Advantage
Today, the Intel x86 is arguable the only chip which retains CISC architecture. This is primarily due to advancements in other areas of computer technology. The price of RAM has decreased dramatically. In 1977, 1MB of DRAM cost about $5,000. By 1994, the same amount of memory cost only $6 (when adjusted for inflation). Compiler technology has also become more sophisticated, so that the RISC use of RAM and emphasis on software has become ideal.

***We have used CISC Architecture for the design of our microprocessor.**

3. REQUIRED TOOLS AND SOFTWARE

3.1 MODELSIM

Mentor Graphics was the first to combine single kernel simulator (SKS) technology with a unified debug environment for Verilog, VHDL, and SystemC. The combination of industry-leading, native SKS performance with the best integrated debug and analysis environment make **ModelSim®** the simulator of choice for both ASIC and FPGA designs. The best standards and platform support in the industry make it easy to adopt in the majority of process and tool flows. To use this software on any PC we require a license (paid). Using this software we can easily simulate and evaluate our design, whether the design is working as desired or not. We can see these results with the help of graphical representations, which is a very user friendly approach. We can vary any parameter according to ou requirement. We have used his software for the simulations of our design, whose screenshots have been attached in the last section of this report.

FEATURES

- **Unified mixed language simulation engine** for ease of use and performance.
- **Native support** of Verilog, SystemVerilog for design, VHDL, and SystemC for effective verification of sophisticated design environments.
- **Fast time-to-debug**, easy to use, multi-language debug environment.
- **Advanced code coverage and analysis** tools for fast time to coverage closure.
- **Interactive and Post-Sim Debug** available so same debug environment used for both.

- **Powerful Waveform** compare for easy analysis of differences and bugs.
- **Unified Coverage Database** with complete interactive and HTML reporting and processing for understanding and debugging coverage throughout your project.
- **Coupled with HDL Designer and HDL Author** for complete design creation, project management and visualization capabilities.

3.2 Hardware Description Language

Hardware description language (HDL) is a specialized computer language used to program electronic and digital logic circuits. The structure, operation and design of the circuits are programmable using HDL. HDL includes a textual description consisting of operators, expressions, statements, inputs and outputs. Instead of generating a computer executable file, the HDL compilers provide a gate map. The gate map obtained is then downloaded to the programming device to check the operations of the desired circuit. The language helps to describe any digital circuit in the form of structural, behavioural and gate level and it is found to be an excellent programming language for FPGAs and CPLDs.

The **three common HDLs are Verilog, VHDL, and SystemC.** Of these, SystemC is the newest. The HDLs will allow fast design and better verification. In most of the industries, Verilog and VHDL are common. Verilog, one of the main Hardware Description Language standardized as IEEE 1364 is used for designing all types of circuits. It consists of modules and the language allows Behavioral, Dataflow and Structural Description. VHDL (Very High Speed Integrated Circuit Hardware Description Language) is standardized by IEEE1164. The design is composed of entities consisting of multiple architectures. SystemC is a language that consist a set of C++classes and macros. It allows electronic system level and transaction modelling.

Need for HDLs

The Moore's Law in the year 1970 has brought a drastic change in the field of IC technology. This change has made the developers to bring out complex digital and electronic circuits. But the problem was the absence of a better programming language allowing hardware and software co design. Complex digital circuit designs require more time for development, synthesis, simulation and debugging. The arrival of HDLs has helped to solve this problem by allowing each module to be worked by a separate team.
All the goals like power, throughput, latency (delay), test coverage, functionality and area consumption required for a design can be known by using HDL. As a result, the designer can make the necessary engineering trade-offs and can develop the design in a better and efficient way. Simple syntax, expressions, statements, concurrent and sequential programming is also necessary while describing the electronics circuits. All these features can be obtained by using a hardware description language.

MODELLING

In structural data flow modelling, digital design functions are defined using components such as an invertor, a MUX, a adder, a decoder, basic digital logic gates etc. It is like connecting and arranging different parts of circuits available to implement a functions you are looking for.

Implementation of half adder using structural data flow modelling:

```
input A, B;
output sum, carry;
xor (.in1(A), .in2(B), .out(sum));
and (.in1(A), .in2(B), .out(carry));
```

Behavioural data flow modelling is used to describe the behaviour of digital circuits. Designer describes the functionality of design by writing algorithm kind of code. It is used to explain flow of data using various Boolean expression such as & (and), |(or), ~(invert), ^(xor), ?(conditional) etc... two sub-categories of this type modelling scheme is continuous data flow and procedural data flow.

1. Continuous data flow

In this type of data flow modelling, assignment of data to outputs are continuous. This will be implemented using "*assign*" statements in Verilog as shown in example.

```
input A, B;
output sum, carry;
wire sum, carry;
assign sum = A ^ B;
assign carry = A & B;
```

It is continuous assignment and hence any change in input will effect output immediately because absence of controlling actions i.e. sensitivity list. Here, outputs variable must be defined as "*wire*" type because they are continuously driven by digital logic defined on RHS.

2. Procedural data flow

Here the data assignments are not carried out continuously instead it happens on specific events specified in sensitivity list. This type of modelling scheme is implemented using procedural blocks such as "*always*" or "*initial*" as shown in example.

```
input A, B;
output sum, carry;
reg sum, carry;
always @ (A)
begin
sum <= A + B;
carry <= A & B;
end
```

The RHS of all expressions is computed continuously but its assignment to LHS happens only on any change in input A but not B. In this case, output variables must be defined as "*reg*" type because they keeps holding previous value until new assignment occurs after any change in specified sensitivity list.

3.3 VERILOG

Verilog, standardized as IEEE 1364, is a hardware description language (HDL) used to model electronic systems.

Hardware description languages such as Verilog differ from software programming languages because they include ways of describing the propagation time and signal strengths (sensitivity). There are two types of assignment operators; a blocking assignment (=), and a non-blocking (<=) assignment. The non-blocking assignment allows designers to describe a state-machine update without needing to declare and use temporary storage variables. Since these concepts are part of Verilog's language semantics, designers could quickly write descriptions of large circuits in a relatively compact and concise form.

Like C, Verilog is case-sensitive and has a basic preprocessor (though less sophisticated than that of ANSI C/C++). Its control flow keywords (if/else, for, while, case, etc.) are equivalent, and its operator precedence is compatible with C. Syntactic differences include: required bit-widths for variable declarations, demarcation of procedural blocks (Verilog uses begin/end instead of curly braces {}), and many other minor differences. Verilog requires that variables be given a definite size.

A Verilog design consists of a hierarchy of modules. Modules encapsulate design hierarchy, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks, and instances of other modules (sub-hierarchies). Sequential statements are placed inside a begin/end block and executed in sequential order within the block. However, the blocks themselves are executed concurrently, making Verilog a dataflow language.

Verilog's concept of 'wire' consists of both signal values (4-state: "1, 0, floating, undefined") and signal strengths (strong, weak, etc.). This system allows abstract modelling of shared signal lines, where multiple sources drive a common net. When a wire has multiple drivers, the wire's (readable) value is resolved by a function of the source drivers and their strengths.

4. Design of the microprocessor in modules

4.1 MEMORY

4.1.1 RAM

The memories we have studied so far can all be read and written. Such memories are called **RAM**s (Random Access Memories), which is a misnomer because all memory chips are randomly accessible, but the term is too well established to get rid of now. RAMs come in two varieties, static and dynamic.

Static RAMs (**SRAM**s), are constructed internally using circuits similar to our basic D flip-flop. These memories have the property that their contents are retained as long as the power is kept on: seconds, minutes, hours, even days. Static RAMs are very fast. A typical access times is a few nsec. For this reason, static RAMS are popular as level 2 cache memory.

Dynamic RAMs (**DRAM**s), in contrast, do not use flip-flops. Instead, a dynamic RAM is an array of cells, each cell containing one transistor and a tiny capacitor. The capacitors can be charged or discharged, allowing 0s and 1s to be stored. Because the electric charge tends to leak out, each bit in a dynamic RAM must be **refreshed** (reloaded) every few milliseconds to prevent the data from leaking away. Because external logic must take care of the refreshing, dynamic RAMs require more complex interfacing than static ones, although in many applications this disadvantage is compensated for by their larger capacities. Since dynamic RAMs need only one transistor and one capacitor per bit (vs. six transistors per bit for the best static RAM), dynamic RAMs have a very high density (many bits per chip). For this reason, main memories are nearly always built out of dynamic RAMs. However, this large capacity has a price: dynamic RAMs are slow (tens of nanoseconds). Thus the combination of a static RAM cache and a dynamic RAM main memory attempts to combine the good properties of each.

4.1.2 ROM

In many applications, such as toys, appliances, and cars, the program and some of the data must remain stored even when the power is turned off. Furthermore, once installed, neither the program nor the data are ever changed. These requirements have led to the development of **ROM**s (Read-Only Memories), which cannot be changed or erased, intentionally or otherwise. The data in a ROM are inserted during its manufacture, essentially by exposing a photosensitive material through a mask containing the desired bit pattern and then etching away the exposed (or unexposed) surface. The only way to change the program in a ROM is to replace the entire chip. ROMs are much cheaper than RAMs when ordered in large enough volumes to defray the cost of making the mask. However, they are inflexible, because they cannot be changed after manufacture, and the turnaround time between placing an order and receiving the ROMs may be weeks. To make it easier for companies to develop new ROM-based products, the **PROM** (Programmable ROM) was invented. A PROM is like a ROM, except that it can be programmed (once) in the field, eliminating the turnaround time. Many PROMs contain an array of tiny fuses inside. A specific fuse can be blown out by selecting its row and column and then applying a high voltage to a special pin on the chip.

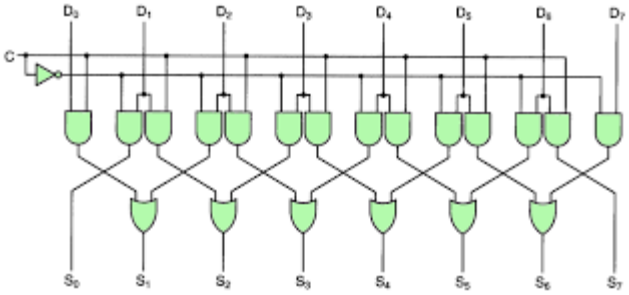
The next development in this line was the **EPROM** (Erasable PROM), which can be not only field-programmed but also field-erased. When the quartz window in an EPROM is exposed to a strong ultraviolet light for 15 minutes, all the bits are set to 1. If many changes are expected during the design cycle, EPROMs are far more economical than PROMs because they can be reused. EPROMs usually have the same organization as static RAMs.

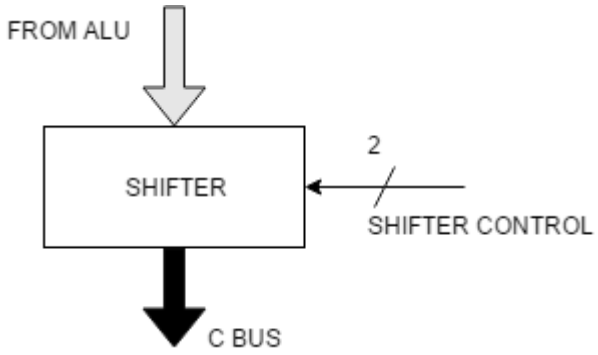
Even better than the EPROM is the **EEPROM** which can be erased by applying pulses to it instead of requiring it to be put in a special chamber for exposure to ultraviolet light. In addition,

an EEPROM can be reprogrammed in place whereas an EPROM has to be inserted in a special EPROM programming device to be programmed.
A more recent kind of EEPROM is **flash memory**. Unlike EPROM, which is erased by exposure to ultraviolet light, and EEPROM, which is byte erasable, flash memory is block erasable and rewritable. Like EEPROM, flash memory can be erased without removing it from the circuit.

4.2 SHIFTER

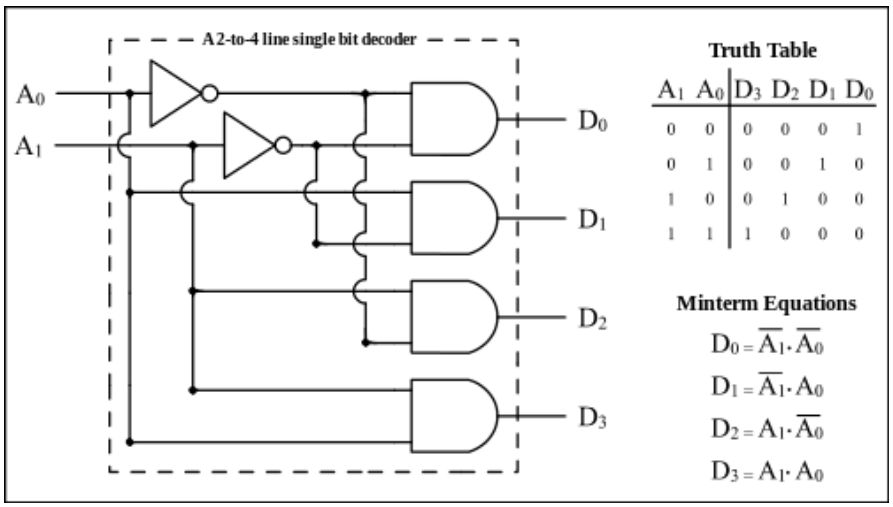
Our first arithmetic MSI circuit is an eight-input, eight-output shifter. Eight bits of input are presented on lines *D*0 , ..., *D* 7 . The output, which is just the input shifted 1 bit, is available on lines *S*0, ..., *S* 7 . The control line, *C*, determines the direction of the shift, 0 for left and 1 for right. On a left shift, a 0 is inserted into bit 7. Similarly, on a right shift, a 1 is inserted into bit 0. To see how the circuit works, notice the pairs of AND gates for all the bitsexcept the gates on the end. When *C* = 1, the right member of each pair is turned on, passing the corresponding input bit to output. Because the right AND gate is wired to the input of the OR gate to its right, a right shift is performed. When *C* = 0, it is the left member of the AND gate pair that turns on, doing a left shift.

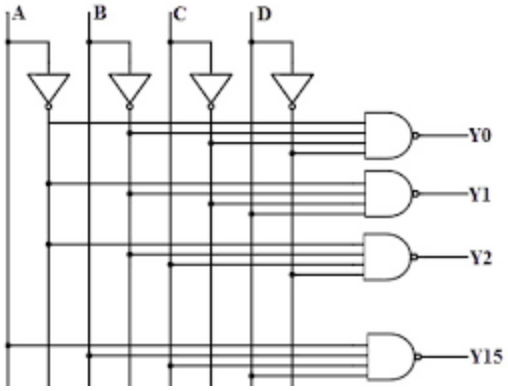




4.3 DECODER

In both the multiplexer and the de-multiplexer, part of the circuits *decode* the address inputs, i.e. it translates a binary number of n digits to 2^noutputs, one of which (the one that corresponds to the value of the binary number) is 1 and the others of which are 0.
It is sometimes advantageous to separate this function from the rest of the circuit, since it is useful in many other applications. Thus, we obtain a new combinatorial circuit that we call the *decoder*.





4-16 DECODER

4.4 ALU

An arithmetic logic unit (ALU) is a combinational digital electronic circuit that performs arithmetic and bitwise operations on integer binary numbers. This is in contrast to a floating-point unit (FPU), which operates on floating point numbers. An ALU is a fundamental building block of many types of computing circuits, including the central processing unit (CPU) of computers, FPUs, and graphics processing units (GPUs). A single CPU, FPU or GPU may contain multiple ALUs.

The inputs to an ALU are the data to be operated on, called operands, and a code indicating the operation to be performed and, optionally, status information from a previous operation; the ALU's output is the result of the performed operation. In many designs, the ALU also exchanges additional information with a status register, which relates to the result of the current or previous operations.

SIGNALS

An ALU has a variety of input and output nets, which are the electrical conductors used to convey digital signals between the ALU and external circuitry. When an ALU is operating, external circuits apply signals to the ALU inputs and, in response, the ALU produces and conveys signals to external circuitry via its outputs.

DATA

A basic ALU has three parallel data buses consisting of two input operands (A and B) and a result output (Y). Each data bus is a group of signals that conveys one binary integer number.

Typically, the A, B and Y bus widths (the number of signals comprising each bus) are identical and match the native word size of the external circuitry (e.g., the encapsulating CPU or other processor).

OPCODE

The opcode input is a parallel bus that conveys to the ALU an operation selection code, which is an enumerated value that specifies the desired arithmetic or logic operation to be performed by the ALU. The opcode size (its bus width) is related to the number of different operations the ALU can perform; for example, a four-bit opcode can specify up to sixteen different ALU operations. Generally, an ALU opcode is not the same as a machine language opcode, though in some cases it may be directly encoded as a bit field within a machine language opcode.

STATUS

The status outputs are various individual signals that convey supplemental information about the result of an ALU operation. These outputs are usually stored in registers so they can be used in future ALU operations or for controlling conditional branching. The collection of bit registers that store the status outputs are often treated as a single, multi-bit register, which is referred to as the "status register" or "condition code register". General-purpose ALUs commonly have status signals such as:

Carry-out, which conveys the carry resulting from an addition operation, the borrow resulting from a subtraction operation, or the overflow bit resulting from a binary shift operation.

Zero, which indicates all bits of output are logic zero.

Negative, which indicates the result of an arithmetic operation is negative.

Overflow, which indicates the result of an arithmetic operation has exceeded the numeric range of output.

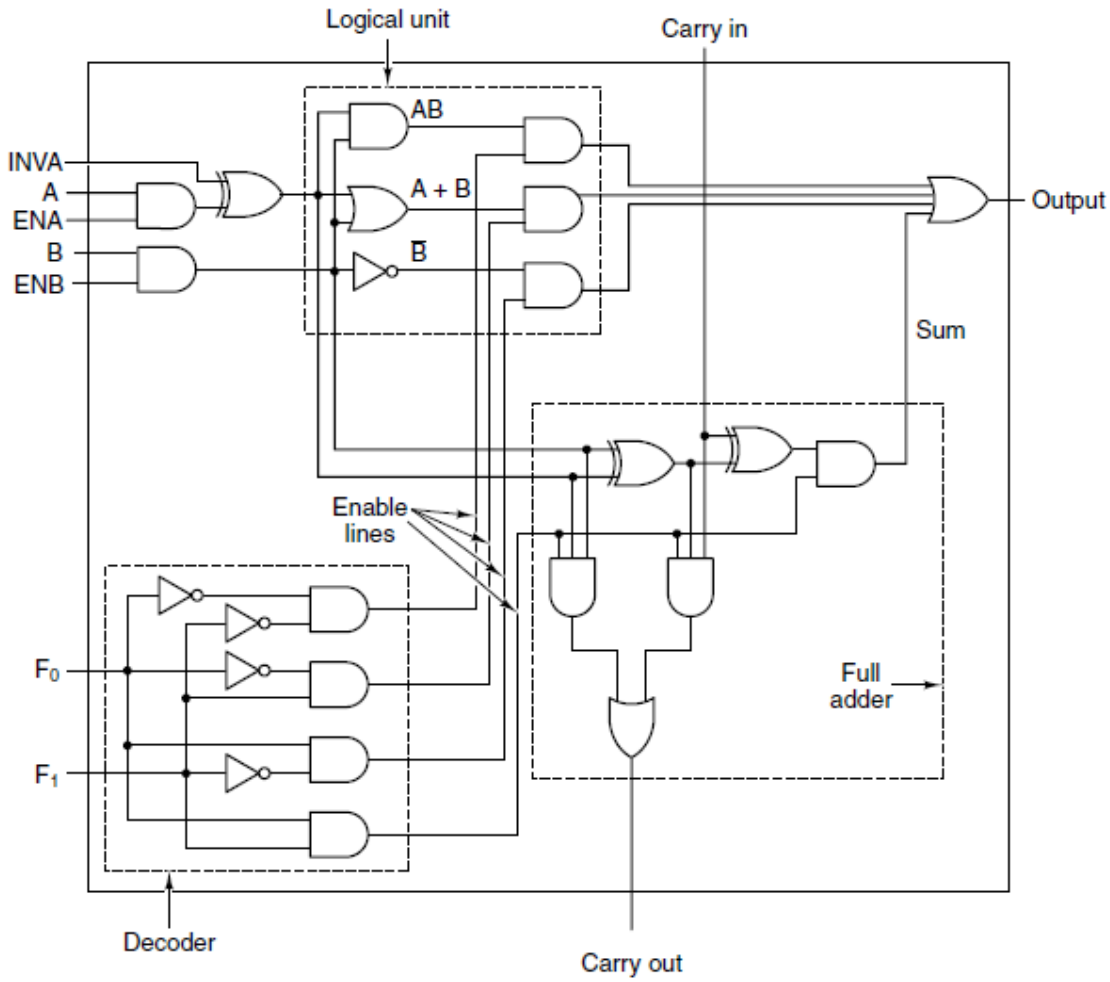
Parity, which indicates whether an even or odd number of bits in the output are logic one.

4.4.1 CIRCUIT OPERATION

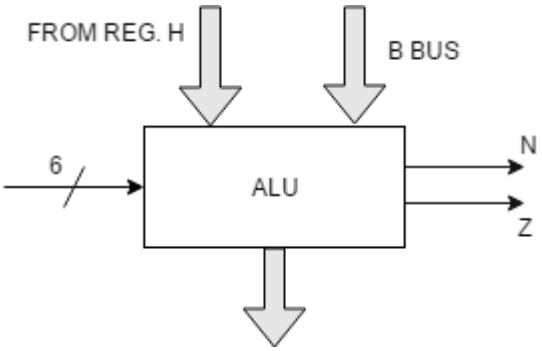
An ALU is a combinational logic circuit, meaning that its outputs will change asynchronously in response to input changes. In normal operation, stable signals are applied to all of the ALU inputs and, when enough time (known as the "propagation delay") has passed for the signals to propagate through the ALU circuitry, the result of the ALU operation appears at the ALU outputs. The external circuitry connected to the ALU is responsible for ensuring the stability of ALU input signals throughout the operation, and for allowing sufficient time for the signals to propagate through the ALU before sampling the ALU result.

In general, external circuitry controls an ALU by applying signals to its inputs. Typically, the external circuitry employs sequential logic to control the ALU operation, which is paced by a clock signal of a sufficiently low frequency to ensure enough time for the ALU outputs to settle under worst-case conditions.

For example, a CPU begins an ALU addition operation by routing operands from their sources (which are usually registers) to the ALU's operand inputs, while the control unit simultaneously applies a value to the ALU's opcode input, configuring it to perform addition. At the same time, the CPU also routes the ALU result output to a destination register that will receive the sum. The ALU's input signals, which are held stable until the next clock, are allowed to propagate through the ALU and to the destination register while the CPU waits for the next clock. When the next clock arrives, the destination register stores the ALU result and, since the ALU operation has completed, the ALU inputs may be set up for the next ALU operation.



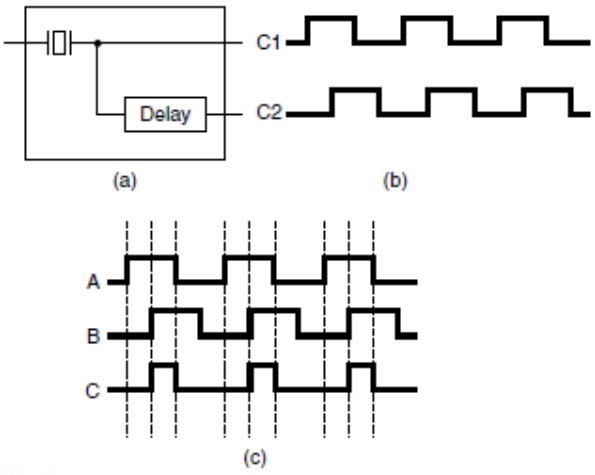
1 bit ALU



4.5 CLOCKS

In many digital circuits the order in which events happen is critical. Sometimes one event must precede another, sometimes two events must occur simultaneously. To allow designers to achieve the required timing relations, many digital circuits use clocks to provide synchronization. A **clock** in this context is a circuit that emits a series of pulses with a precise pulse width and precise interval between consecutive pulses. The time interval between the corresponding edges of two consecutive pulses is known as the **clock cycle time**. Pulse frequencies are commonly between 1 and 500 MHz, corresponding to clock cycles of 1000 nsec to 2 nsec. To achieve high accuracy, the clock frequency is usually controlled by a crystal oscillator. In a computer, many events may happen during a single clock cycle. If these events must occur in a specific order, the clock cycle must be divided into subcycles. A common way of providing finer resolution than the basic clock is to tap the primary clock line and insert a

circuit with a known delay in it, thus generating a secondary clock signal that is phase-shifted from the primary, as shown in Fig. (a). The timing diagram of Fig. (b) provides four time references for
discrete events:
1. Rising edge of C1.
2. Falling edge of C1.
3. Rising edge of C2.
4. Falling edge of C2.
By tying different events to the various edges, the required sequencing can be achieved. If more than four time references are needed within a clock cycle, more secondary lines can be tapped from the primary, with different delays.



(a) A clock. (b) The timing diagram for the clock. (c) Generation of an asymmetric clock.

4.6 REGISTERS

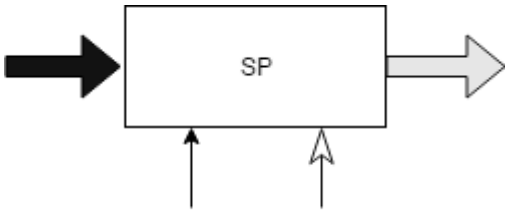
ISA-level registers can be roughly divided into two categories: special purpose registers and general-purpose registers. The special-purpose registers include things like the program counter and stack pointer, as well as other registers with a specific function. In contrast, the general-purpose registers are there to hold key local variables, and intermediate results of calculations. Their main function is to provide rapid access to heavily-used data (basically, avoiding memory accesses).On some machines, the general-purpose registers are completely symmetric and interchangeable. The choice of register does not matter. However, on other machines some of the general-purpose registers may be somewhat special.
In addition to the ISA-level registers visible to user programs, there are always a substantial number of special-purpose registers available only in kernel mode. These registers control the various caches, memory, I/O devices, and other hardware features of the machine. They are used only by the operating system, so compilers and users do not have to know about them.

PSW (Program Status Word)
One control register that is something of a kernel/user hybrid is the flags register or PSW. This register holds various miscellaneous bits that are needed by the CPU. The most important bits are the condition codes. These bits are set on every ALU cycle and reflect the status of the result of the most recent operation. Typical condition code bits include N — Set when the result was Negative. Z — Set when the result was Zero. V — Set when the result caused an overflow. C — Set when the result caused a Carry out of the leftmost bit. A — Set when there was a carry out of

bit 3 (Auxiliary carry). P — Set when the result had even Parity. The condition codes are important because the comparison and conditional branch instructions (i.e., conditional jump instructions) use them.

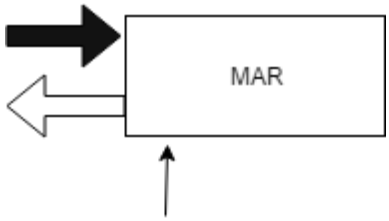
STACK POINTER (SP)

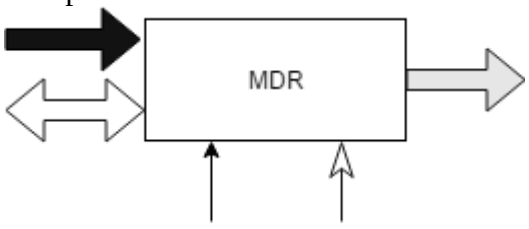
When a procedure is called, the register stack pointer is advanced so the input parameters are visible in registers, but no registers are allocated for local variables. The procedure itself decides how many registers it needs and advances the register stack pointer to allocate them. These registers need not be saved on entry or restored on exit although if the procedure needs to modify a static register it must take care to explicitly save it first and restore it later. The code that saves the old frame pointer, sets up the new frame pointer, and advances the stack pointer to reserve space for local variables is called the procedure prolog. The stack pointer, which points to the top of the stack



Memory Address Register (MAR) & Memory Data Register (MDR)

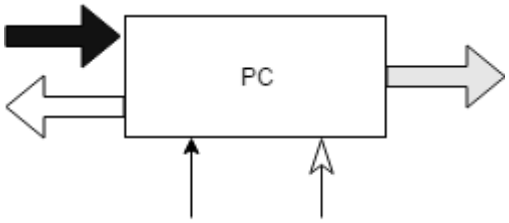
Our machine has two different ways to communicate with memory: a 32-bit, word-addressable memory port and an 8-bit, byte-addressable memory port. The 32-bit port is controlled by two registers, MAR (Memory Address Register), and MDR (Memory Data Register). The 8-bit port is controlled by one register, PC, which reads 1 byte into the low-order 8 bits of MBR. This port can only read data from memory; it cannot write data to memory. Each of these registers (and all the other registers) are driven by one or two control signals. An open arrow under a register indicates a control signal that enables the register's output onto the B bus. Since MAR does not have a connection to the B bus, it does not have an enable signal. H does not have one either because it is always enabled, it being the only possible left ALU input. MAR contains word addresses, so that the values 0, 1, 2, etc., refer to consecutive words. PC contains byte addresses, so that the values 0, 1, 2, etc. refer to consecutive bytes. Putting a 2 in MAR and starting a memory read will read out bytes 8–11 (i.e., word 2) from memory and put them in MDR





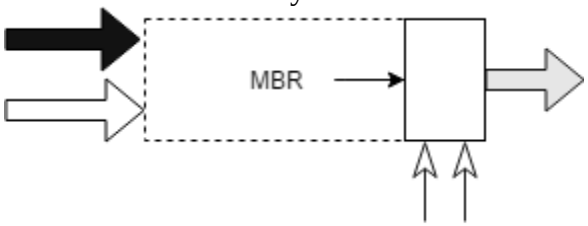
PROGRAM COUNTER (PC)

The most important register is the Program Counter (PC), which points to the next instruction to be fetched for execution. (The name ‘‘program counter’’ is somewhat misleading because it has nothing to do with counting anything, but the term is universally used. Also important is the Instruction Register (IR), which holds the instruction currently being executed. (Most computers have numerous other registers as well, some of them general purpose as well as some for specific purposes.



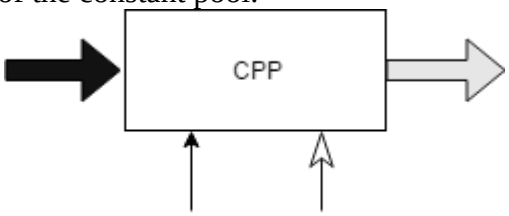
Memory Byte Register (MBR)

Data read from memory through the 8-bit memory port are returned in MBR, an 8-bit register. MBR can be gated (i.e., copied) onto the B bus in one of two ways: unsigned or signed. When the unsigned value is needed, the 32-bit word put onto the B bus contains the MBR value in the low-order 8 bits and zeros in the upper 24 bits. Unsigned values are useful for indexing into a table, or when a 16-bit integer has .The choice of whether the 8-bit MBR is converted to an unsigned or a signed 32-bit value on the B bus is determined by which of the two control signals is asserted.



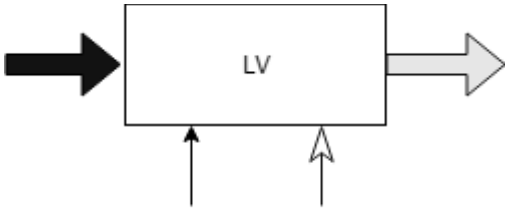
Constant Pool Pointer (CPP)

The Constant Pool. This area cannot be written by an IJVM program and consists of constants, strings, and pointers to other areas of memory that can be referenced. It is loaded when the program is brought into memory and not changed afterward. There is an implicit register, CPP that contains the address of the first word of the constant pool.



Local Variable (LV)

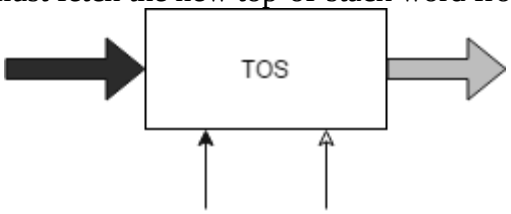
An area of memory, called the stack, is reserved for variables, but individual variables do not get absolute addresses in it. Instead, a register, say, LV, is set to point to the base of the local variables for the current procedure. A procedure A, which has local variables a1, a2, and a3, has been called, so storage for its local variables has been reserved starting at the memory location pointed to by LV. Another register, SP, points to the highest word of A's local variables. If LV is 100 and words are 4 bytes, then SP will be 108. Variables are referred to by giving their offset (distance) from LV. The data structure between LV and SP (and including both words pointed to) is called A's local variable frame.



Top of Stack (TOS)

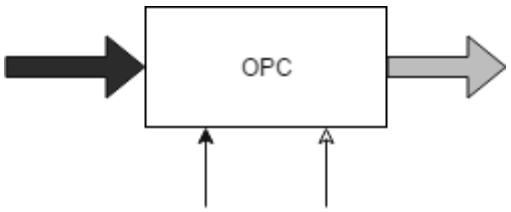
At the beginning and end of each instruction, TOS contains the value of the memory location pointed to by SP, the top word on the stack. This value is redundant since it can always be read from memory, but having it in a register often saves a memory reference. For a few instructions

maintaining TOS means more memory operations. For example, the POP instruction throws away the top word, and therefore must fetch the new top-of-stack word from the memory into TOS.

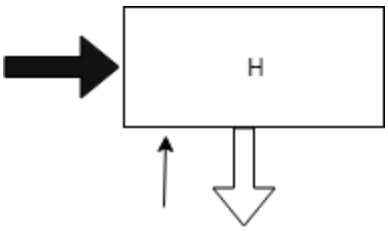


Old Program Counter(OPC)

The OPC register is a temporary (i.e., scratch) register. It has no preassigned use. It is used, for example, to save the address of the opcode for a branch instruction while PC is incremented to access parameters. It is also used as a temporary register in the IJVM conditional branch instructions.



Holding Register



4.7 AN EXAMPLE MICRO ARCHITECTURE

Why IJVM instead of SUN JVM?

Our microarchitecture will contain a microprogram (in ROM), whose job is to fetch, decode, and execute IJVM instructions. We cannot use the Sun JVM interpreter for the microprogram because we need a tiny microprogram that drives the individual gates in the actual hardware efficiently. In contrast, the Sun JVM interpreter was written in C for portability, and cannot control the hardware at the level of detail we need.

A convenient model for the design of the microarchitecture is to think of the design as a programming problem, where each instruction at the ISA level is a function to be called by a master program. In this model, the master program is a simple, endless loop that determines a function to be invoked, calls the function, then starts over.

The microprogram has a set of variables, called the state of the computer, which can be accessed by all the functions. Each function changes at least some of the variables making up the state. Here, a subset of the Java Virtual Machine has been chosen. This subset contains only integer instructions, so we have named it IJVM. IJVM has some complex instructions. IJVM instructions are short and sweet. Each instruction has a few fields, usually one or two, each of which has some specific purpose. The first field of every instruction is the opcode (short for operation code), which identifies the instruction, telling whether it is an ADD or a BRANCH, or something else. Many

instructions have an additional field, which specifies the operand. For example, instructions that access a local variable need a field to tell which variable. This model of execution, sometimes called the fetch-execute cycle, is useful in the abstract and may also be the basis for implementation for ISAs like IJVM that have complex instructions.

4.8 DATA PATHS

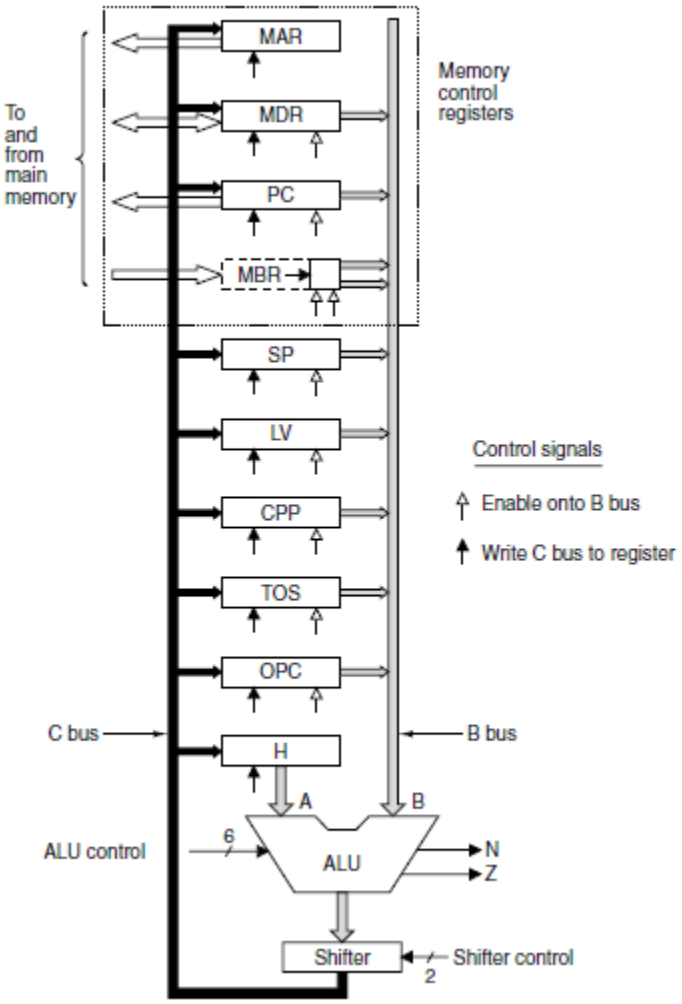
The data path is that part of the CPU containing the ALU, its inputs, and its outputs. It contains a number of 32-bit registers, to which we have assigned symbolic names such as PC, SP, and MDR. They are given these names because they usually hold a value corresponding to the variable of the same name in the ISA level architecture. Most registers can drive their contents onto the B bus. The output of the ALU drives the shifter and then the C bus, whose value can be written into one or more registers at the same time. The functions of ALU is determined by six control lines. The short diagonal line labelled ‘‘6’’, indicates that there are six ALU control lines. These are F0 and F1 for determining the ALU operation, ENA and ENB for individually enabling the inputs, INVA for inverting the left input, and INC for forcing a carry into the low-order bit, effectively adding 1 to the result.

F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\overline{A}$
1	0	1	1	0	0	$\overline{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	0	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	–1

Useful combinations of ALU signals and the function performed.

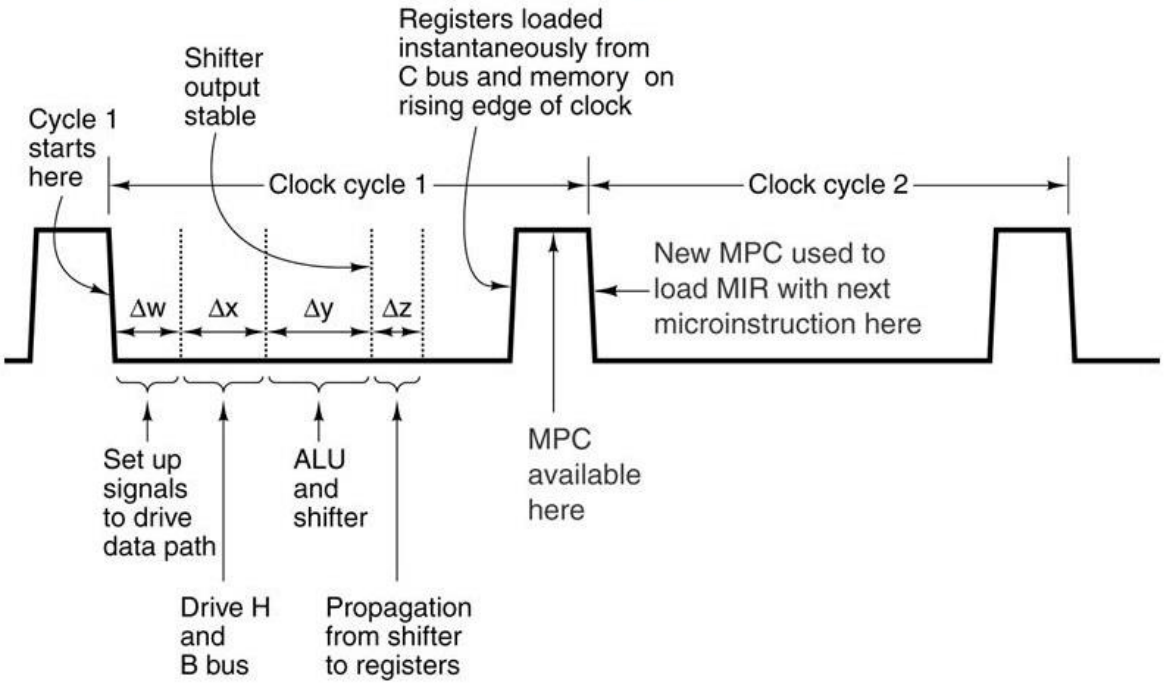
The ALU needs two data inputs: a left input (A) and a right input (B). Attached to the left input is a holding register, H. Attached to the right input is the B bus, which can be loaded from any one of nine sources, indicated by the nine gray arrows touching it. H can be loaded by choosing an ALU function that just passes the right input (from the B bus) through to the ALU output. One such function is adding the ALU inputs, only with ENA negated so the left input is forced to zero. Adding zero to the value on the B bus just yields the value on the B bus. This result can then be passed through the shifter unmodified and stored in H. In addition to the above functions, two other control lines can be used independently to control the output from the ALU. SLL8 (Shift Left Logical) shifts the contents left by 1 byte, filling the 8 least significant bits with zeros. SRA1 (Shift Right Arithmetic) shifts the contents right by 1 bit, leaving the most significant bit unchanged. It is explicitly possible to read and write the same register on one cycle. How can a

register be read and written on the same cycle without producing garbage? The solution is that reading and writing are
actually performed at different times within the cycle. When a register is selected as the ALU's right input, its value is put onto the B bus early in the cycle and kept there continuously throughout the entire cycle. The ALU then does its work, producing a result that passes through the shifter onto the C bus. Near the end of the cycle, when the ALU and shifter outputs are known to be stable, a clock signal triggers the store of the contents of the C bus into one or more of the registers. One of these registers may well be the one that supplied the B bus with its input. The precise timing of the data path makes it possible to read and write the same register on one cycle,



DATA PATH OF MICRO ARCHITECTURE

Data Path Timing



The largest and most important item in the control portion of the machine is a memory called the control store. It is convenient to think of it as a memory that holds the complete microprogram, although it is sometimes implemented as a set of logic gates. In general, we will refer to it as the control store, to avoid confusion with the main memory, accessed through MBR and MDR. However, functionally, the control store is a memory that simply holds microinstructions instead of ISA instructions. For our example machine, it contains 512 words, each word consisting of one 36-bit microinstruction of the kind.

Actually, not all of these words are needed, but (for reasons to be explained shortly) we need addresses for 512 distinct words. In one important way, the control store is quite different from the main memory: instructions in main memory are always executed in address order (except for branches); microinstructions are not. The act of incrementing the program counter expresses the fact that the default instruction to execute after the current one is the instruction following the current one in memory. Microprograms need more flexibility (because microinstruction sequences tend to be short), so they usually do not have this property. Instead, each microinstruction explicitly specifies its successor.

Since the control store is functionally a (read-only) memory, it needs its own memory address register and its own memory data register. It does not need read and write signals, because it is continuously being read. We will call the control store's memory address register MPC (MicroProgram Counter). This name is ironic since the locations in it are explicitly not ordered, so the concept of counting is not useful (but who are we to argue with tradition?). The memory data register is called MIR (MicroInstruction Register). Its function is to hold the current microinstruction, whose bits drive the control signals that operate the data .

In principle, we could describe the control store in binary, 36 bits per word. But as in conventional programming languages, there is great benefit in introducing notation that conveys the essence of

the issues we need to deal with while obscuring the details that can be ignored, or can be better handled automatically. It is important to realize here that the language we have chosen is intended to illustrate the concepts rather than to facilitate efficient designs. If the latter were our goal, we would use a different notation to maximize the flexibility available to the designer.

Here a short pulse is produced at the start of each clock cycle. It can be derived from the main clock, as shown in figure. On the falling edge of the pulse, the bits that will drive all the gates are set up. This takes a finite and known time, Δw. Then the register needed on the B bus is selected and driven onto the B bus. It takes Δx before the value is stable. Then the ALU and shifter begin to operate on valid data. After another Δy, the ALU and shifter outputs are stable. After an additional Δz, the results have propagated along the C bus to the registers, where they can be loaded on the rising edge of the next pulse. The load should be edge triggered and fast, so that even if some of the input registers are changed, the effects will not be felt on the C bus until long after the registers have been loaded. Also on the rising edge of the pulse, the register driving the B bus stops doing so, in preparation for the next cycle. It is important to realize that even though there are no storage elements in the data path, there is a finite propagation time through it. Changing the value on the B bus does not cause the C bus to change until a finite time later (due to the finite delays of each step). Consequently, even if a store changes one of the input registers, the value will be safely tucked away in the register long before the (nowincorrect) value being put on the B bus (or H) can reach the ALU.
Making this design work requires rigid timing, a long clock cycle, a known minimum propagation time through the ALU, and a fast load of the registers from the C bus. A somewhat different way to look at the data path cycle is to think of it as broken up into implicit subcycles. The start of subcycle 1 is triggered by the falling edge of the clock. The activities that go on during the subcycles are shown below along with the subcycle lengths (in parentheses).

1. The control signals are set up (Δw).
2. The registers are loaded onto the B bus (Δx).
3. The ALU and shifter operate (Δy).
4. The results propagate along the C bus back to the registers (Δz).

At the rising edge of the next clock cycle, the results are stored in the registers.

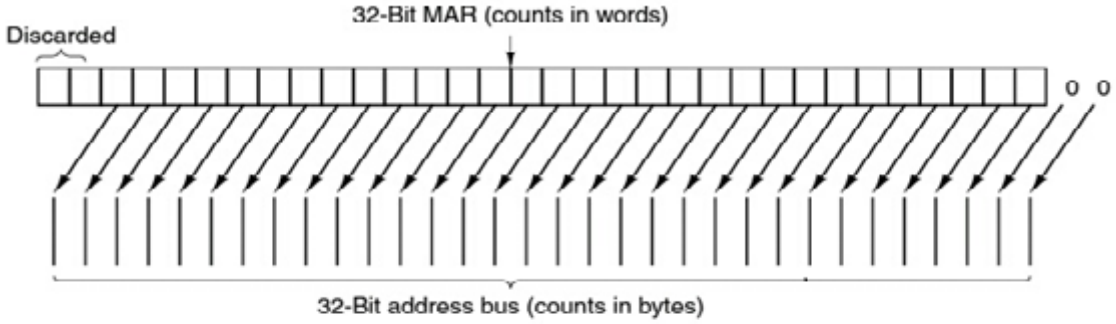
In reality, the ALU and shifter run all the time. However, their inputs are garbage until a time $\Delta w + \Delta x$ after the falling edge of the clock. Likewise, their outputs are garbage until $\Delta w + \Delta x + \Delta y$ has elapsed after the falling edge of the clock. The only explicit signals that drive the data path are the falling edge of the clock, which starts the data path cycle, and the rising edge of the clock, which loads the registers from the C bus. The other subcycle boundaries are implicitly determined by the inherent propagation times of the circuits involved.

4.10 MEMORY OPERATION

Our machine has two different ways to communicate with memory: a 32-bit, word-addressable memory port and an 8-bit, byte-addressable memory port. The 32-bit port is controlled by two registers, MAR (**Memory Address Register**), and MDR (**Memory Data Register**). Each of these registers (and all the other registers in Fig. 4-1) are driven by one or two **control signals**. An open arrow under a register indicates a control signal that enables the register's output onto the B bus. Since MAR does not have
a connection to the B bus, it does not have an enable signal. H does not have one either because it is always enabled, it being the only possible left ALU input. A solid black arrow under a register indicates a control signal that writes (i.e., loads) the register from the C bus. Since MBR cannot be loaded from the C bus, it does not have a write signal (although it does have two other enable signals, described below). To initiate a memory read or write, the appropriate memory registers

must be loaded, then a read or write signal issued to the memory. MAR contains *word* addresses, so that the values 0, 1, 2, etc., refer to consecutive words. PC contains *byte* addresses, so that the values 0, 1, 2, etc. refer to consecutive
bytes. Thus putting a 2 in PC and starting a memory read will read out byte 2 from memory and put it in the low-order 8 bits of MBR. Putting a 2 in MAR and starting a memory read will read out bytes 8–11 (i.e., word 2) from memory and put them in MDR. This difference in functionality is needed because MAR and PC will be used to reference two different parts of memory. For the moment, suffice it to say that the MAR/MDR combination is used to read and write ISA-level data words and the PC/MBR combination is used to read the executable ISA-level program, which consists of a byte stream.

In the actual physical implementation, there is only one real memory and it is byte oriented. Allowing MAR to count in words (needed due to the way JVM is defined) while the physical memory counts in bytes is handled by a simple trick. When MAR is placed on the address bus, its 32 bits do not map onto the 32 address lines, 0 – 31, directly. Instead MAR bit 0 is wired to address bus line 2, MAR bit 1 is wired to address bus line 3, and so on. The upper 2 bits of MAR are discarded since they are only needed for word addresses above 2^{32}, none of which are legal for our 4-GB machine. Using this mapping, when MAR is 1, address 4 is put onto the bus; when MAR is 2, address 8 is put onto the bus, and so forth.

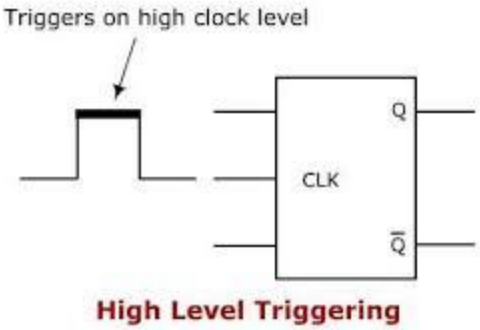


Mapping of the bits in MAR to the address bus.

As mentioned above, data read from memory through the 8-bit memory port are returned in MBR, an 8-bit register. MBR can be gated (i.e., copied) onto the B bus in one of two ways: unsigned or signed. When the unsigned value is needed, the 32-bit word put onto the B bus contains the MBR value in the low-order 8 bits and zeros in the upper 24 bits. Unsigned values are useful for indexing into a table, or when a 16-bit integer has to be assembled from 2 consecutive (unsigned) bytes in the instruction stream. The other option for converting the 8-bit MBR to a 32-bit word is to treat it as a signed value between −128 and +127 and use this value to generate a 32-bit word with the same numerical value. This conversion is done by duplicating the MBR sign bit (leftmost bit) into the upper 24 bit positions of the B bus, a process known as **sign extension**. When this option is chosen, the upper 24 bits will either be all 0s or all 1s, depending on whether the leftmost bit of the 8-bit MBR is a 0 or a 1. The choice of whether the 8-bit MBR is converted to an unsigned or a signed 32-bit value on the B bus is determined by which of the two control signals is asserted. The need for these two options is why two arrows are present. Types of interrupt:

Level-triggered

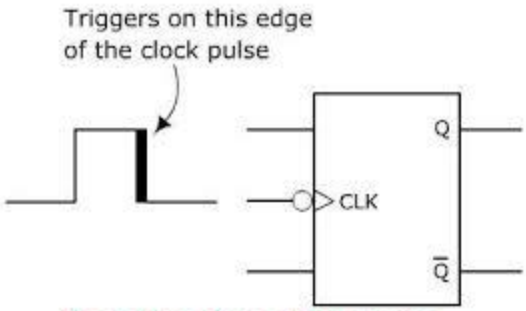
A level-triggered interrupt is an interrupt signalled by maintaining the interrupt line at a high or low level. A device wishing to signal a Level-triggered interrupt drives the interrupt request line to its active level (high or low), and then holds it at that level until it is serviced. It ceases asserting the line when the CPU commands it to or otherwise handles the condition that caused it to signal the interrupt.



High Level Triggering

Edge-triggered

An edge-triggered interrupt is an interrupt signalled by a level transition on the interrupt line, either a falling edge (high to low) or a rising edge (low to high). A device, wishing to signal an interrupt, drives a pulse onto the line and then releases the line to its inactive state. If the pulse is too short to be detected by polled I/O then special hardware may be required to detect the edge.



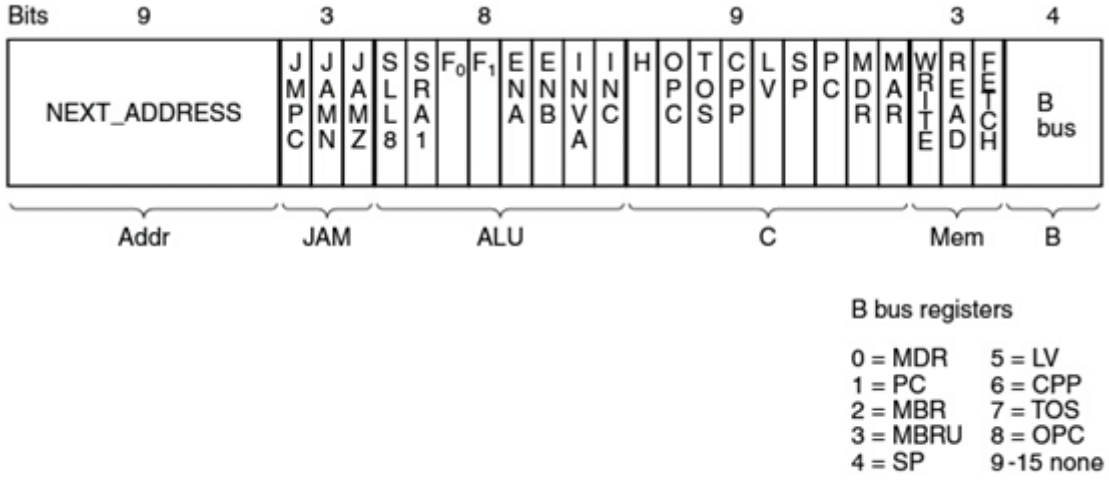
Negative Edge Triggering

4.11MICROINSTRUCTIONS

To control the data path, we need 29 signals. These can be divided into five functional groups, as described below:

9 Signals to control writing data from the C bus into registers.
9 Signals to control enabling registers onto the B bus for ALU input.
8 Signals to control the ALU and shifter functions.
2 Signals (not shown) to indicate memory read/write via MAR/MDR.
1 Signal (not shown) to indicate memory fetch via PC/MBR.

The values of these 29 control signals specify the operations for one cycle of the data path. A cycle consists of gating values out of registers and onto the B bus, propagating the signals through the ALU and shifter, driving them onto the C bus, and finally writing the results in the appropriate register or registers. In addition, if a memory read data signal is asserted, the memory operation is started at the end of the data path cycle, after MAR has been loaded. The memory data are available at the very end of the *following* cycle in MBR or MDR and can be used in the cycle *after that*. In other words, a memory read on either port initiated at the end of cycle *k* delivers data that cannot be used in cycle $k + 1$, but only in cycle
$k + 2$ or later.

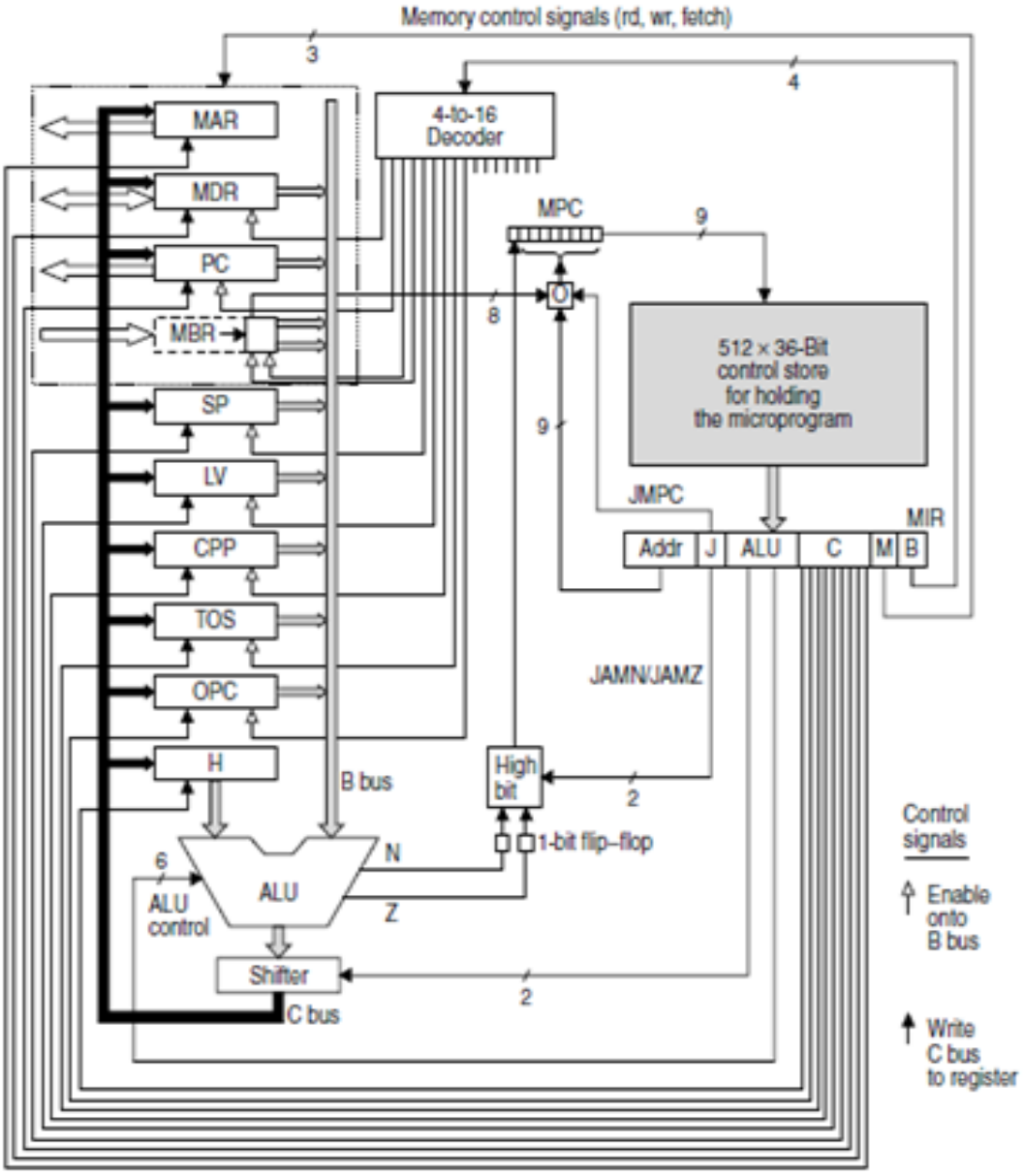


The microinstruction format

we can control the data path with 9 + 4 + 8 + 2 + 1 = 24 signals, hence 24 bits. However, these 24 bits only control the data path for one cycle. The second part of the control is to determine what is to be done on the following cycle. To include this in the design of the controller, we will create a format for describing the operations to be performed using the 24 control bits plus two additional fields: the NEXT"ADDRESS field and the JAM field. A possible format, divided into the six groups, and containing the following 36 signals is considered:

Addr – Contains the address of a potential next microinstruction.
JAM – Determines how the next microinstruction is selected.
ALU – ALU and shifter functions.
C – Selects which registers are written from the C bus.
Mem – Memory functions.
B – Selects the B bus source

4.12 MICRO INSTRUCTION CONTROL



Here it is decided which of the control signals should be enabled on each cycle. This is determined by a **sequencer** that is responsible for stepping through the sequence of operations necessary for the execution of a single ISA instruction. The sequencer must produce two kinds of information each cycle:
1. The state of every control signal in the system.
2. The address of the microinstruction that is to be executed next.
The block diagram has two parts: the data path, on the left, and the control section, on the right.

CONTROL STORE
The largest and most important item in the control portion of the machine is a memory called the **control store**. It is convenient to think of it as a memory that holds the complete microprogram, although it is sometimes implemented as a set of logic gates. In general, we will refer to it as the control store, to avoid confusion with the main memory, accessed through MBR and MDR. However, functionally, the control store is a memory that simply holds microinstructions instead

of ISA instructions. Since the control store is functionally a (read-only) memory, it needs its own memory address register and its own memory data register. It does not need read and write signals, because it is continuously being read. We will call the control store's memory address register **MPC** (**MicroProgram Counter**). This name is ironic since the locations in it are explicitly not ordered, so the concept of counting is not useful (but who are we to argue with tradition?). The memory data register is called **MIR** (**MicroInstruction Register**). Its function is to hold the current microinstruction, whose bits drive the control signals that operate the data path.

The MIR register holds the same six groups. The Addr and J (for JAM) groups control the selection of the next microinstruction and will be discussed shortly. The ALU group contains the 8 bits that select the ALU function and drive the shifter. The C bits cause individual registers to load the ALU output from the C bus. The M bits control memory operations. Finally, the last 4 bits drive the decoder that determines what goes onto the B bus. In this case we have chosen to use a standard 4-to-16 decoder, even though only nine possibilities are required. In a more finely-tuned design, a 4-to-9 decoder could be used. The trade-off here is using a standard circuit taken from a library of circuits versus designing a custom one.

OPERATION

At the start of each clock cycle, MIR is loaded from the word in the control store pointed to by MPC. The MIR load time is indicated in the figure by Δw. If one thinks in terms of subcycles, MIR is loaded during the first one. Once the microinstruction is set up in MIR, the various signals propagate out into the data path. A register is put out onto the B bus, the ALU knows which operation to perform, and there is lots of activity out there. This is the second subcycle. After an interval $\Delta w + \Delta x$ from the start of the cycle, th0e ALU inputs are stable.
Another Δy later, everything has settled down and the ALU, N, Z, and shifter outputs are stable. The N and Z values are then saved in a pair of 1-bit flip-flops. These bits, like all the registers that are loaded from the C bus and from memory, are saved on the rising edge of the clock, near the end of the data path cycle. The ALU output is not latched but just fed into the shifter. The ALU and shifter activity occurs during subcycle 3. After an additional interval, Δz, the shifter output has reached the registers via the C bus. Then the registers can be loaded near the end of the cycle. Subcycle 4 consists of loading the registers and N and Z flip-flops. It terminates a little after the rising edge of the clock, when all the results have been saved and the results of the previous memory operations are available and MPC has been loaded. This process goes on and on until somebody gets bored with it and turns the machine off.

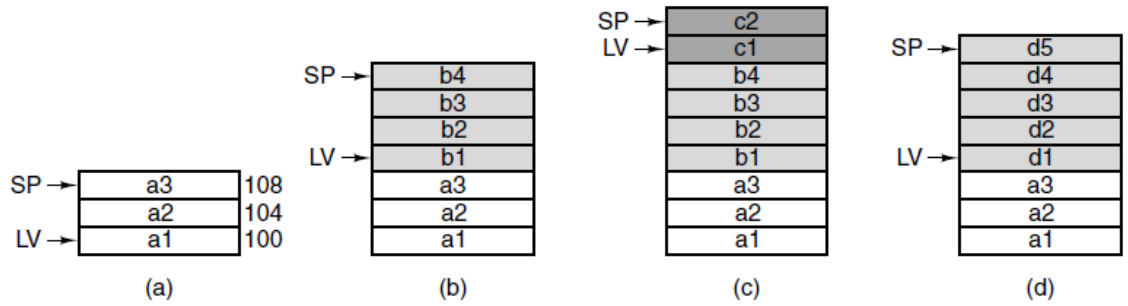
In parallel with driving the data path, the microprogram has to determine which microinstruction to execute next, as they need not be run in the order they appear in the control store. The calculation of the address of the next microinstruction begins after MIR has been loaded and is stable. First, the 9-bit NEXT"ADDRESS field is copied to MPC. While this copy is taking place, the JAM field is inspected. If it has the value 000, nothing else is done; when the copy of NEXT"ADDRESS completes, MPC will point to the next microinstruction. If one or more of the JAM bits are 1, more work is needed. If JAMN is set, the 1-bit N flip-flop is ORed into the high-order bit of MPC. Similarly, if JAMZ is set, the 1-bit Z flip-flop is ORed there. If both are set, both are ORed there. The reason that the N and Z flip-flops are needed is that after the rising edge of the clock (while the clock is high), the B bus is no longer being driven, so the ALU outputs can no longer be assumed to be correct. Saving the ALU status flags in N and Z makes the correct values available and stable for the MPC computation, no matter what is going on around the ALU.

the logic that does this computation is labeled ‘‘High bit.’’ The Boolean function it computes is
F = (JAMZ AND Z) OR (JAMN AND N) OR NEXT"ADDRESS[8]
Note that in all cases, MPC can take on only one of two possible values:

1. The value of NEXT"ADDRESS.
2. The value of NEXT"ADDRESS with the high-order bit ORed with 1.

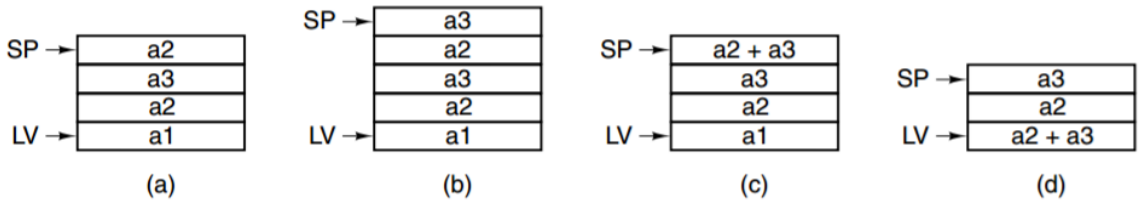
STACKS

An area of memory, called the **stack**, is reserved for variables, but individual variables do not get absolute addresses in it. Instead, a register, say, LV, is set to point to the base of the local variables for the current procedure. In Fig. (a), a procedure *A*, which has local variables *a1*, *a2*, and *a3*, has been called, so storage for its local variables has been reserved starting at the memory location pointed to by LV. Another register, SP, points to the highest word of *A*'s local variables. If LV is 100 and words are 4 bytes, then SP will be 108. Variables are referred to by giving their offset (distance) from LV. The data structure between LV and SP (and including both words pointed to) is called *A*'s **local variable frame**.



Use of a stack for storing local variables. (a) While *A* is active. (b) After *A* calls *B*. (c) After *B* calls *C*. (d) After *C* and *B* return and *A* calls *D*.

Now let us consider what happens if *A* calls another procedure, *B*. Where should *B*'s four local variables (*b1, b2, b3, b4*) be stored? Answer: On the stack, on top of *A*'s, as shown in Fig. (b). Notice that LV has been adjusted by theprocedure call to point to *B*'s local variables instead of *A*'s. *B*'s local variables can be referred to by giving their offset from LV. Similarly, if *B* calls *C*, LV and SP are adjusted again to allocate space for *C*'s two variables, as shown in Fig. (c). When *C* returns, *B* becomes active again, and the stack is adjusted back to Fig. (b) so that LV now points to *B*'s local variables again. Likewise, when *B* returns, we get back to the situation of Fig. (a). Under all conditions, LV points to the base of the stack frame for the currently active procedure, and SP points to the top of the stack frame. Now suppose that *A* calls *D*, which has five local variables. We get the situation of Fig. (d), in which *D*'s local variables use the same memory that *B*'s did, as well as part of *C*'s. With this memory organization, memory is only allocated for procedures that are currently active. When a procedure returns, the memory used by its local variables is released.



Use of an operand stack for doing an arithmetic computation.

Stacks have another use, in addition to holding local variables. They can be used for holding operands during the computation of an arithmetic expression. When used this way, the stack is referred to as the **operand stack**. Suppose, for example, that before calling *B*, *A* has to do the computation a1 = a2 + a3;
One way of doing this sum is to push *a2* onto the stack, as shown in Fig. (a). Here SP has been incremented by the number of bytes in a word, say, 4, and the first operand stored at the address now pointed to by SP. Next, *a3* is pushed onto the stack, as shown in Fig. (b). As an aside on notation, we will typeset all program fragments in Helvetica, as above. We will also use this font for assembly language opcodes and machine registers, but in running text, program variables and procedures will be given in *italics*. The difference is that variables and procedure names are chosen by the user; opcodes and register names are built in. The actual computation can be done by now executing an instruction that pops two words off the stack, adds them together, and pushes the result back onto the stack, as shown in Fig. (c). Finally, the top word can be popped off the stack and stored back in local variable *a1*, as illustrated in Fig. (d). The local variable frames and the operand stacks can be intermixed.

4.13 IJVM MEMORY MODEL

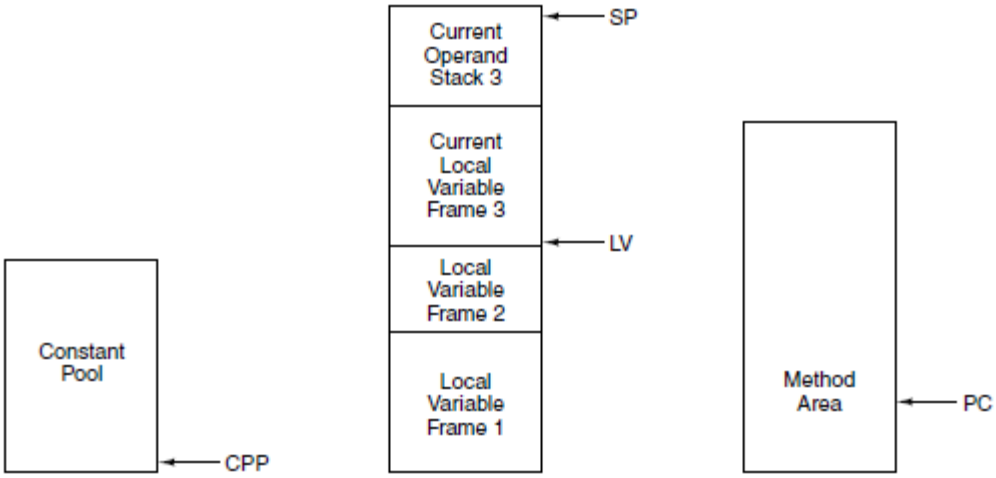
It consists of a memory that can be viewed in two ways- an array of 4294967296 bytes or an array of 1073741824 words each consisting of 4 byes. IJVM instruction can access memory by indexing from these pointers. Following memory areas ca be identified-

1. CONSTANT POOL-This area cannot be written by IJVM program and consists of constants strings and pointers to other areas of memory and not changed afterwards. There is an implicit register CPP, that contains the address of the first word of the constant pool.

2. LOCAL VARIABLE-an area allocated for storing variables during the lifetime of the invocation. It is called the local variable frame. At the beginning of this resides the parameter with which the method was invoke. It does not include the operand stack. However f the operand stack or efficiency sake we implement the operand stack is register is called LV immediately above the local variable frame. An implicit register contains the address off the first location in local variable frame. The parameters passed at the invocation of the method are stored at the beginning of the LV frame.

3. THE OPERAND STACK-The stack frame is guaranteed not to exceed a certain size, computed in advance by the Java compiler. The operand stack space is allocated directly above the local variable frame. There is an implicit register that contains the address of the top word of the stack. Unlike CPP and LV, this pointer, SP, changes during the execution of the method as operands are pushed onto the stack or popped from it.

4. THE METHOD AREA- There is a region of memory containing the program, referred to as the ‘‘text’’ area in a UNIX process. There is an implicit register that contains the address of the instruction to be fetched next. This pointer is referred to as the Program Counter, or PC. Unlike the other regions of memory, the Method Area is treated as a byte array.



The various parts of the IJVM memory.

One point needs to be made regarding the pointers. The CPP, LV, and SP registers are all pointers to *words*, not *bytes*, and are offset by the number of words. For the integer subset we have chosen, all references to items in the constant pool, the local variables frame, and the stack are words, and all offsets used to index
into these frames are word offsets. In contrast, PC contains a byte address, and an addition or subtraction to PC changes the address by a number of bytes, not a number of words. Addressing for PC is different from the others.

4.14 The IJVM Instruction Set

Instruction set or instruction set architecture is the structure of the computer that provides commands to the computer to guide the computer for processing data manipulation. Instruction set consists of instructions, addressing modes, native data types, registers, interrupt, exception handling and memory architecture. Instruction set can be emulated in software by using an interpreter or built into hardware of the processor. Instruction Set Architecture can be considered as a boundary between the software and hardware. Classification of microprocessors can be done based on the RISC and CISC instruction set architecture.

Instruction set specifies processor functionality including the operations supported by the processor, storage mechanisms of the processor, and the way of compiling the programs to the processor.

Each instruction consists of an opcode and sometimes an operand, such as a memory offset or a constant. The first column gives the hexadecimal encoding of the instruction. The second gives its assembly language mnemonic. The third gives a brief description of its effect.

HEX	MNEMONIC	MEANING
0x10	BIPUSH *byte*	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ *offset*	Pop word from stack and branch if it is zero

0x9B	IFELT *offset*	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ *offset*	Pop two words from stack; branch if equal
0x84	IINC *varnum const*	Add a constant to the local variable
0x15	ILOAD *varnum*	Push local variable onto stack
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W *index*	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word from top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

*In above table, the operands byte, const, and varnum are 1 byte. The operands disp, index and offset are 2 byte.

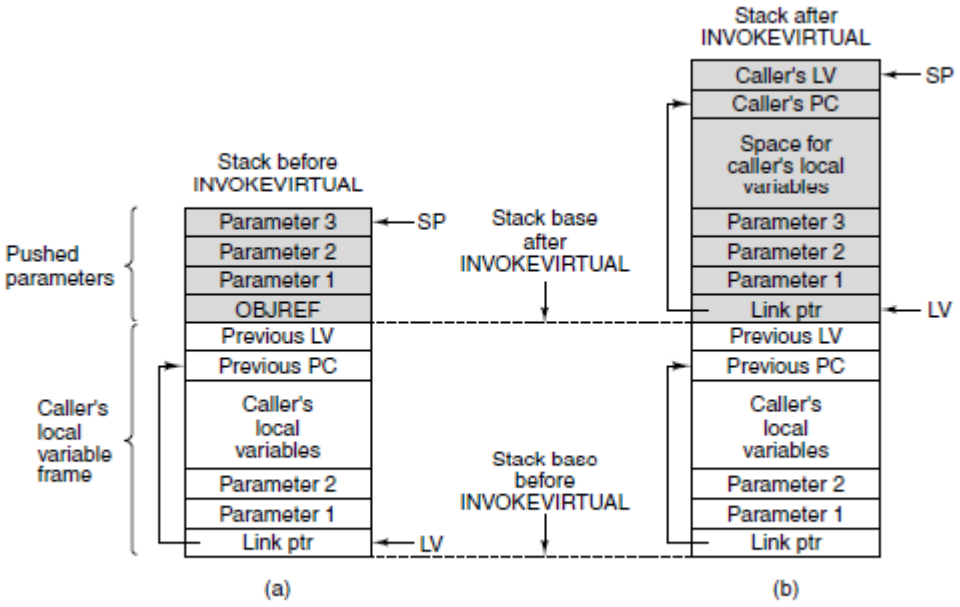
1. INVOKEVIRTUAL-

There is an instruction (INVOKEVI RTUAL) for invoking another method, and another instruction (IRETURN) for exiting the method and returning control to the method that invoked it. Due to the complexity of the mechanism we have slightly simplified the definition, making it possible to produce a straightforward mechanism for invoking a call and return. The restriction is that, unlike Java, we only allow a method to invoke a method existing within its own object. This restriction severely cripples the object orientation but allows us to present a much simpler mechanism, by avoiding the requirement to locate the method dynamically. On all computers except JVM, the address of the procedure to call is determined directly by the CALL instruction, so our approach is actually the normal case, not the exception. The mechanism for invoking a method is as follows. First, the caller pushes onto the stack a reference (pointer) to the object to be called. (This reference is not needed in IJVM since no other object may be specified, but it is retained for consistency with JVM.) Then the caller pushes the method's parameters onto the stack, in this example,Parameter 1, Parameter 2, and Parameter 3. Finally, INVOKEVIRTUAL is executed.The INVOKEVIRTUAL instruction includes a displacement which indicates the position in the constant pool that contains the start address within the Method Area for the method being invoked. However, while the method code resides at the location pointed to by this pointer, the first 4 bytes in the method area contain special data. The first 2 bytes are interpreted as a 16-bit integer indicating the number of parameters for the method (the parameters themselves have previously been pushed onto the stack). For this count, OBJREF is counted as a parameter parameter 0. This 16-bit integer, together with the value of SP, provides the location of OBJREF. The second 2 bytes in the method area are interpreted as another 16-bit integer indicating the size of the local variable area for the method being invoked. This is necessary because a new stack will be established for the method, beginning immediately above the local variable frame. Finally, the fifth byte in the method area contains the first opcode to be executed.

The actual sequence that occurs for INVOKEVIRTUAL is as follows-

The two unsigned index bytes that follow the opcode are used to construct an index into the constant pool table (the first byte is the high-order byte). The instruction computes the base address of the new local variable frame by subtracting off the number of parameters from the stack pointer and setting LV to point to OBJREF. At this location, overwriting OBJREF, the implementation stores the address of the location where the old PC is to be stored. This address is computed by adding the size of the local variable frame (parameters + local variables) to the address contained in LV. Immediately above the address where the old PC is to be stored is the address where the old LV is to be stored. Immediately above that address is the beginning of the stack for the newly-

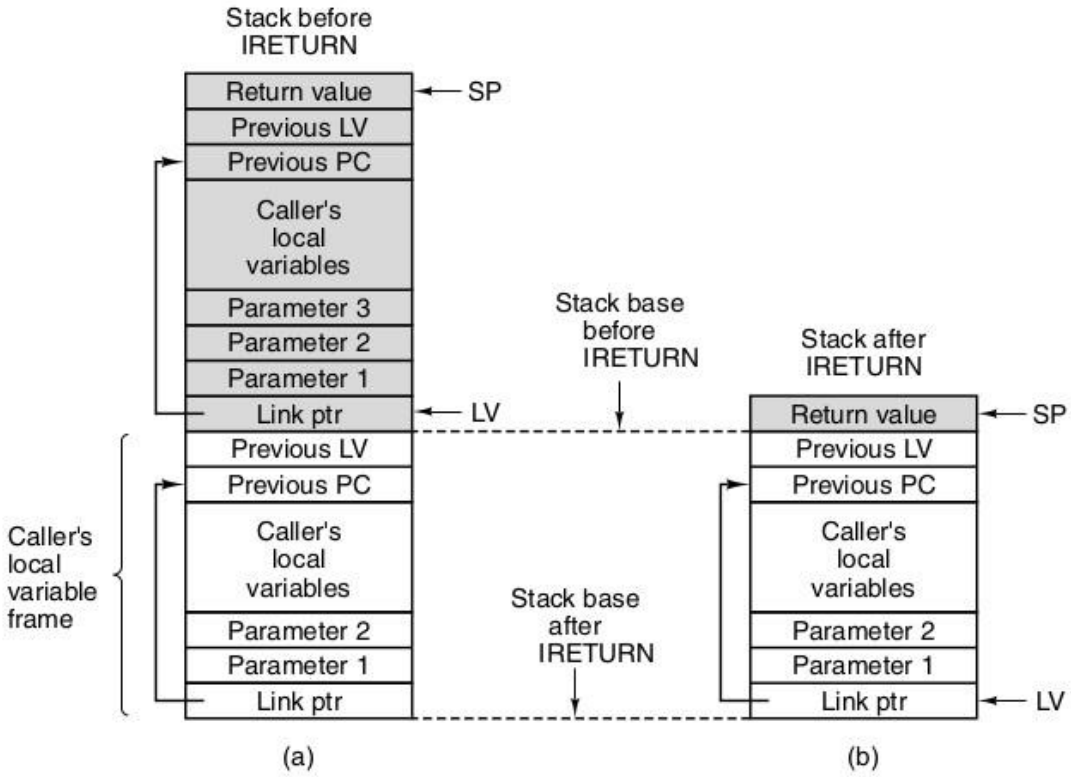
called procedure. SP is set to point to the old LV, which is the address immediately below the first empty location on the stack.000 SP always points to the top word on the stack. If the stack is empty, it points to the first location below the end of the stack because our stacks grow upward, toward higher addresses. In our figures, stacks always grow upward, toward the higher address at the top of the page.The last operation needed to carry out INVOKEVIRTUAL is to set PC to point to the fifth byte in the method code space.



(a) Memory before executing INVOKEVIRTUAL. (b) After executing it.

2. IRETURN -

The IRETURN instruction reverses the operations of the INVOKEVIRTUAL instruction. It deallocates the space used by the returning method. It also restores the stack to its former state, except that (1) the (now overwritten) OBJREF word and all the parameters have been popped from the stack, and (2) the returned value has been placed at the top of the stack, at the location formerly occupied by OBJREF. To restore the old state, the IRETURN instruction must be able to restore the PC and LV pointers to their old values. 2It does this by accessing the link pointer (which is the word identified by the current LV pointer). In this location, remember, where the OBJREF was originally stored, the INVOKEVIRTUAL instruction stored the address containing the old PC. This word and the word above it are retrieved to restore PC and LV, respectively, to their old values. The return value, which is stored at the top of the stack of the terminating method, is copied to the location where the OBJREF was originally stored, and SP is restored to point to this location. Control is therefore returned to the instruction immediately following the INVOKEVIRTUAL instruction.



(a)Memory before executing IRETURN. (b)After executing it.

4.15 AN EXAMPLE IMPLEMENTATION

Having specified both the microarchitecture and the macro architecture in detail, the remaining issue is the implementation.

Microinstructions and Notation

In principle, we could describe the control store in binary, 36 bits per word. But as in conventional programming languages, there is great benefit in introducing notation that conveys the essence of the issues we need to deal with while obscuring the details that can be ignored, or can be better handled automatically. It is important to realize here that the language we have chosen is intended to illustrate the concepts rather than to facilitate efficient designs. If the latter were our goal, we would use a different notation to maximize the flexibility available to the designer. One aspect where this issue is important is the choice of addresses.

Since the memory is not logically ordered, there is no natural ‘‘next instruction’’ to be implied as we specify a sequence of operations. Much of the power of this control organization derives from the ability of the designer (or the assembler) to select addresses efficiently. We therefore begin by introducing a simple symbolic language that fully describes each operation without explaining fully how all addresses may have been determined. Our notation specifies all the activities that occur in a single clock cycle in a single line. We could, in theory, use a high-level language to describe the operations. However, cycle-by-cycle control is very important because it gives the opportunity to perform multiple operations concurrently, and it is necessary to be able to analyze each cycle to understand and verify the operations. If the goal is a fast, efficient implementation (other things being equal, fast and efficient is always better than slow and inefficient), then every cycle counts. In a real implementation, many subtle tricks are hidden in the program, using obscure sequences or operations in order to save a single cycle. There is a high payoff for saving cycles: a four-cycle instruction that can be reduced by two cycles now runs twice as fast. And this speedup is obtained every time we execute the instruction. One possible approach is simply to list the

signals that should be activated each clock cycle. Suppose that in one cycle we want to increment the value of SP. We also want to initiate a read operation, and we want the next instruction to be the one residing at location 122 in the control store. We might write ReadRegister = SP, ALU = INC, WSP, Read, NEXT"ADDRESS = 122 where WSP means ‘‘write the SP register.’’ This notation is complete, but hard to understand. Instead we will combine the operations in a natural and intuitive way to capture the effect of what is happening:

SP = SP + 1; rd

Let us call our high-level Micro Assembly Language ‘‘MAL’’ (French for ‘‘sick,’’ something you become if you have to write too much code in it). MAL is tailored to reflect the characteristics of the microarchitecture. During each cycle, any of the registers can be written, but typically only one is. Only one register can be gated to the B side of the ALU. On the A side, the choices are +1, 0, −1, and the register H. Thus we can use a simple assignment statement, as in Java, to indicate the operation to be performed. For example, to copy something from SP to MDR, we can say

MDR = SP

To indicate the use of the ALU functions other than passing through the B bus, we can write, for example,

MDR = H + SP

which adds the contents of the H register to SP and writes the result into MDR.

The + operator is commutative (which means that the order of the operands does not matter), so the above statement can also be written as

MDR = SP + H

and generate the same 36-bit microinstruction, even though strictly speaking H must be the left ALU operand. We have to be careful to use only legal operations. The most important legal operations are shown in the Fig. 4-16, where SOURCE can be any of MDR, PC, MBR, MBRU, SP, LV, CPP, TOS, or OPC (MBRU implies the unsigned version of MBR). These registers can all act as sources to the ALU on the B bus. Similarly, DEST can be any of MAR, MDR, PC, SP, LV, CPP, TOS, OPC, or H, all of which are possible destinations for the ALU output on the C bus. This format is deceptive because many seemingly reasonable statements are illegal. For example,

MDR = SP + MDR

looks perfectly reasonable, but there is no way to execute it on the data path of Fig. 4-6 in one cycle. This restriction exists because for an addition (other than increment or decrement) one of the operands must be the H register. Likewise,

H = H − MDR

might be useful, but it, too, is impossible, because the only possible source of a subtrahend (the value being subtracted) is the H register. It is up to the assembler to reject statements that look valid but are, in fact, illegal. We extend the notation to permit multiple assignments by the use of multiple equal signs. For example, adding 1 to SP and storing it back into SP as well as writing it into MDR can be accomplished by

SP = MDR = SP + 1

To indicate memory reads and writes of 4-byte data words, we will just put rd and wr in the microinstruction. Fetching a byte through the 1-byte port is indicated by fetch. Assignments and memory operations can occur in the same cycle. This is indicated by writing them on the same line. To avoid any confusion, let us repeat the fact that the Mic-1 has two ways of accessing memory. Reads and writes of 4-byte data words use MAR/MDR and are indicated in the microinstructions by rd and wr, respectively. Reads of 1-byte opcodes from the instruction stream use PC/MBR and are indicated by fetch in the microinstructions. Both kinds of memory operations can proceed simultaneously. However, the same register may not receive a value from memory and the data path in the same cycle. Consider the code

MAR = SP; rd

MDR = H

The effect of the first microinstruction is to assign a value from memory to MDR at the end of the second microinstruction. However, the second microinstruction also assigns a value to MDR at the same time. These two assignments are in conflict and are not permitted as the results are undefined. Remember that each microinstruction must explicitly supply the address of the next microinstruction to be executed. However, it commonly occurs that a microinstruction is invoked only by one other microinstruction, namely, by the one on the line immediately above it. To ease the microprogrammer's job, the microassembler normally assigns an address to each microinstruction (not necessarily consecutive in the control store), and fills in the NEXT"ADDRESS field so that microinstructions written on consecutive lines are executed consecutively. However, sometimes the microprogrammer wants to branch away, either unconditionally or conditionally. The notation for unconditional branches is easy:

goto *label*

can be included in any microinstruction to explicitly name its successor. For example, most microinstruction sequences end with a return to the first instruction of the main loop, so the last instruction in each such sequence typically includes

goto Main1

Note that the data path is available for normal operations even during a microinstruction that contains a goto. After all, every single microinstruction contains a NEXT"ADDRESS field. All goto does is instruct the microassembler to put a specific value there instead of the address where it has decided to place the microinstruction on the next line. In principle, every line should have a goto statement, only as a convenience to the microprogrammer, when the target address is the next line, it may be omitted. For conditional branches, we need a different notation. Remember that JAMN and JAMZ use the N and Z bits, which are set based on the ALU output. Sometimes it is needed to test a register to see if it is zero, for example. One way to do this would be to run it through the ALU and store it back in itself. Writing

TOS = TOS

looks peculiar, although it does the job (setting the Z flip-flop based on TOS). However, to make microprograms look nicer, we now extend MAL, adding two new imaginary registers, N and Z, which can be assigned to. For example,

Z = TOS

runs TOS through the ALU, thus setting the Z (and N) flip-flops, but it does not do a store into any register. What using Z or N as a destination really does is tell the microassembler to set all the bits in the C field of Fig to 0. The data path executes a normal cycle, with all normal operations allowed, but no registers are written to. Note that it does not matter whether the destination is N or Z; the microinstruction generated by the microassembler is identical. Programmers who intentionally choose the ''wrong'' one should be forced to work on a 4.77 MHz original IBM PC for a week as punishment. The syntax for telling the microassembler to set the JAMZ bit is

if (Z) goto L1; else goto L2

Since the hardware requires these two addresses to be identical in their low-order 8 bits, it is up to the microassembler to assign them such addresses. On the other hand, since *L2* can be anywhere in the bottom 256 words of the control store, the microassembler has a lot of freedom in finding an available pair. Normally, these two statements will be combined, for example,

Z = TOS; if (Z) goto L1; else goto L2

The effect of this statement is that MAL generates a microinstruction in which TOS is run through the ALU (but not stored anywhere) so that its value sets the Z bit. Shortly after Z has been loaded from the ALU condition bit, it is ORed into the high-order bit of MPC, forcing the address of the next microinstruction to be fetched from either *L2* or *L1* (which must be exactly 256 more than *L2*). MPC will

be stable and ready to use for fetching the next microinstruction. Finally, we need a notation for using the JMPC bit. The one we will use is

goto (MBR OR *value*)

This syntax tells the microassembler to use *value* for NEXT"ADDRESS and set the JMPC bit so that MBR is ORed into MPC along with NEXT"ADDRESS. If *value* is 0, which is the normal case, it is sufficient to just write
goto (MBR)
Note that only the low-order 8 bits of MBR are wired to MPC, so the issue of sign extension (i.e., MBR versus MBRU) does not arise here. Also note that the MBR available at the end of the current cycle is the one used. A fetch started in *this* microinstruction is too late to affect the choice of the next microinstruction.

DEST = H
DEST = SOURCE
DEST = $\overline{H}$
DEST = $\overline{SOURCE}$
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE – H
DEST = SOURCE – 1
DEST = –H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = –1

All permitted operations. Any of the above operations may be extended by adding "<< 8" to them to shift the result left by 1 byte. For example, a common operation is H = MBR << 8

5. MODULE DESIGN IN VERILOG

ALU

```
module alu #(parameter WORD_WIDTH = 32) (clk,a,b,control,n,z,out);
 input clk;
 input [WORD_WIDTH-1 : 0] a, b;      //inputs
 input [5:0] control;                //control lines
 output reg n, z;                   //flags n=sign z=zero
 output reg [WORD_WIDTH-1 : 0] out;  //output

 initial //initializing flags ant output to zero
 begin
  n = 0;
  z = 0;
  out = 0;
 end

 always @(posedge clk)
 begin
  if (out == 0)
   z <= 1;
  else
   z <= 0;

  n <= out[WORD_WIDTH-1];
 end

 always @(*)
 begin
  (*full_case,parallel_case*)
  case(control)
   6'b011000 :  out <= a;
   6'b010100 :  out <= b;
   6'b011010 :  out <= ~a;
   6'b101100 :  out <= ~b;
   6'b111100 :  out <= a+b;
   6'b111101 :  out <= a+b+1;
   6'b111001 :  out <= a+1;
   6'b110101 :  out <= b+1;
   6'b111111 :  out <= b-a;
   6'b110110 :  out <= b-1;
   6'b111011 :  out <= -a;
   6'b001100 :  out <= a&b;
   6'b011100 :  out <= a|b;
   6'b010000 :  out <= 0;
   6'b010001 :  out <= 1;
   6'b010010 :  out <= -1;
   default   :  out <= 0;
  endcase
 end
endmodule
```

CONTROL STORE

```
module control_store(clk,address,out);
  input clk;
  input [8:0] address;
  output reg [35:0] out;

  reg [35:0] mem [0:511] ;

  initial
  begin
    out <= 36'bZ;
    $readmemb("control_memory.list", mem); // memory_list is memory file
  end

  always @(negedge clk)
  begin
    out <= mem[address];
  end
endmodule
```

TOS

```
module tos #(parameter WORD_WIDTH = 8) (clk,c_bus,c_write_enable,b_bus,b_read_enable);
  input clk;
  input [WORD_WIDTH-1 : 0] c_bus;
  input b_read_enable,c_write_enable;
  output reg [WORD_WIDTH-1 : 0] b_bus;

  reg [WORD_WIDTH-1 : 0] data;

  initial
  begin
    b_bus = 'bZ;
    data = 0;
  end

  always @ (posedge clk)
  begin
    if (c_write_enable)
      data <= c_bus;
  end

  always @ (negedge clk)
  begin
    if (b_read_enable)
      b_bus <= data;
    else b_bus <= 'bZ;
  end

endmodule
```

CPP

```
module cpp #(parameter WORD_WIDTH = 8) (clk,c_bus,c_write_enable,b_bus,b_read_enable);
  input clk;
  input [WORD_WIDTH-1 : 0] c_bus;
```

```
  input b_read_enable,c_write_enable;
  output reg [WORD_WIDTH-1 : 0] b_bus;

  reg [WORD_WIDTH-1 : 0] data;

  initial
  begin
    b_bus = 'bZ;
    data = 0;
  end

  always @ (posedge clk)
  begin
    if (c_write_enable)
      data <= c_bus;
  end

  always @ (negedge clk)
  begin
    if (b_read_enable)
      b_bus <= data;
    else b_bus <= 'bZ;
  end
endmodule
```

H

```
module h #(parameter WORD_WIDTH = 8) (clk,c_bus,c_write_enable,a_bus);
  input clk;
  input c_write_enable;
  input [WORD_WIDTH-1 : 0] c_bus;
  output [WORD_WIDTH-1 : 0] a_bus;
  reg [WORD_WIDTH-1 : 0] data;

  assign a_bus = data;

  initial data = 0;

  always @ (posedge clk)
  begin
    if (c_write_enable)
      data <= c_bus;
  end
endmodule
```

LV

```
module lv #(parameter WORD_WIDTH = 8) (clk,c_bus,c_write_enable,b_bus,b_read_enable);
  input clk;
  input [WORD_WIDTH-1 : 0] c_bus;
  input b_read_enable,c_write_enable;
  output reg [WORD_WIDTH-1 : 0] b_bus;
```

```
 reg [WORD_WIDTH-1 : 0] data;

 initial
 begin
  b_bus = 'bZ;
  data = 0;
 end

 always @ (posedge clk)
 begin
  if (c_write_enable)
    data <= c_bus;
 end

 always @ (negedge clk)
 begin
  if (b_read_enable)
    b_bus <= data;
  else b_bus <= 'bZ;
 end

endmodule
```

MAR

```
module mar #(parameter WORD_WIDTH = 8) (clk,c_bus,c_write_enable,mem_out);
 input clk;
 input [WORD_WIDTH-1 : 0] c_bus;
 input c_write_enable;
 output  [WORD_WIDTH-1 : 0] mem_out;
 reg [WORD_WIDTH-1 : 0] data;

 assign mem_out = data;

 initial data = 0;

 always @(posedge clk)
 begin
  if (c_write_enable)
    data <= c_bus;
 end
endmodule
```

MBR

```
module mbr (clk,mem,mem_read_enable,b_bus,b_write_enable,data);
 input clk;
 input [7 : 0] mem;
 input [1 : 0] b_write_enable;
 input mem_read_enable;
 output reg [31 : 0] b_bus;
 output reg [7 : 0] data;

 initial
 begin
```

```
  b_bus = 'bZ;
  data = 8'b01100000;
 end

 always @ (posedge clk)
 begin
  if (mem_read_enable)
   data <= mem;
 end

 always @ (negedge clk)
 begin
  if (b_write_enable == 2'b01)
   b_bus <= {24'b0,data};
  else if (b_write_enable == 2'b10)
   b_bus <= {{24{data[7]}},{data[7:0]}};
  else b_bus <= 'bZ;
 end

endmodule
```

MDR

```
Module mdr #(parameter WORD_WIDTH = 8)
(clk,c_bus,c_write_enable,b_bus,b_read_enable,mem,mem_read,mem_write);
 input clk;
 input [WORD_WIDTH-1 : 0] c_bus;
 input b_read_enable,c_write_enable,mem_read,mem_write;
 output reg [WORD_WIDTH-1 : 0] b_bus;
 inout [WORD_WIDTH-1 : 0] mem;

 reg [WORD_WIDTH-1 : 0] data;

 assign mem = mem_write ? data : 'bZ ;

 initial
 begin
  b_bus = 'bZ;
  data = 0;
 end

 always @ (posedge clk)
 begin
  if (c_write_enable)
   data <= c_bus;
  else if (mem_read)
   data <= mem;
 end

 always @ (negedge clk)
 begin
  if (b_read_enable)
   b_bus <= data;
  else b_bus <= 'bZ;
 end
```

```
endmodule
```

OPC

```
module opc #(parameter WORD_WIDTH = 8) (clk,c_bus,c_write_enable,b_bus,b_read_enable);
  input clk;
  input [WORD_WIDTH-1 : 0] c_bus;
  input b_read_enable,c_write_enable;
  output reg [WORD_WIDTH-1 : 0] b_bus;

  reg [WORD_WIDTH-1 : 0] data;

  initial
  begin
    b_bus = 'bZ;
    data = 0;
  end

  always @ (posedge clk)
  begin
    if (c_write_enable)
      data <= c_bus;
  end

  always @ (negedge clk)
  begin
    if (b_read_enable)
      b_bus <= data;
    else b_bus <= 'bZ;
  end

endmodule
```

PC

```
module pc #(parameter WORD_WIDTH = 8)
(clk,c_bus,c_write_enable,b_bus,b_read_enable,mem,mem_read);
  input clk;
  input [WORD_WIDTH-1 : 0] c_bus;
  input b_read_enable,c_write_enable,mem_read;
  output reg [WORD_WIDTH-1 : 0] b_bus;
  output [WORD_WIDTH-1 : 0] mem;

  reg [WORD_WIDTH-1 : 0] data;

  initial
  begin
    b_bus = 'bZ;
    data = 0;
  end

  always @ (posedge clk)
  begin
```

```
    if (c_write_enable)
      data <= c_bus;
    else if (mem_read)
      data <= mem;
  end

  always @ (negedge clk)
  begin
    if (b_read_enable)
      b_bus <= data;
    else b_bus <= 'bZ;
  end
endmodule
```

SP

```
module sp #(parameter WORD_WIDTH = 8) (clk,c_bus,c_write_enable,b_bus,b_read_enable);
  input clk;
  input [WORD_WIDTH-1 : 0] c_bus;
  input b_read_enable,c_write_enable;
  output reg [WORD_WIDTH-1 : 0] b_bus;

  reg [WORD_WIDTH-1 : 0] data;

  initial
  begin
    b_bus = 'bZ;
    data = 0;
  end

  always @ (posedge clk)
  begin
    if (c_write_enable)
      data <= c_bus;
  end

  always @ (negedge clk)
  begin
    if (b_read_enable)
      b_bus <= data;
    else b_bus <= 'bZ;
  end
endmodule
```

DECODER

```
module decoder(in,out);
  input [3:0] in;
  output [15:0] out;
  assign out = 1<<in;
endmodule
```

PROCESSOR

```
module processor(clk,reset,word_address,word_data,byte_address,byte_data,read,write,fetch);
  input clk,reset;
  inout [31:0] word_data;
  input [7:0] byte_data;
  output [31:0] word_address;
  output [31:0] byte_address;
  output read,write,fetch;

  wire [31:0] a_bus,b_bus,c_bus,alu_output;
  wire [35:0] control_signal;
  wire [15:0] decoder_output;
  wire [8:0] control_store_address_bus;
  wire [7:0] mbr_data;
  wire N,Z,high_bit;

  assign high_bit = (control_signal[24] & Z) | (control_signal[25] & N);
  assign control_store_address_bus[8] = high_bit | control_signal[35];
  assign control_store_address_bus[7:0] = (control_signal[26] == 1)? mbr_data : control_signal[34:27];

  mar #(32) MAR(clk,c_bus,control_signal[7],word_address);
  mdr #(32)
MDR(clk,c_bus,control_signal[8],b_bus,decoder_output[0],word_data,control_signal[5],control_signal[6
]);
  pc #(32) PC(clk,c_bus,control_signal[9],b_bus,decoder_output[1],byte_address,control_signal[5]);
  mbr MBR(clk,byte_data,control_signal[4],b_bus,decoder_output[3:2],mbr_data);
  sp #(32) SP(clk,c_bus,control_signal[10],b_bus,decoder_output[4]);
  lv #(32) LV(clk,c_bus,control_signal[11],b_bus,decoder_output[5]);
  cpp #(32) CPP(clk,c_bus,control_signal[12],b_bus,decoder_output[6]);
  tos #(32) TOS(clk,c_bus,control_signal[13],b_bus,decoder_output[7]);
  opc #(32) OPC(clk,c_bus,control_signal[14],b_bus,decoder_output[8]);
  h #(32) H(clk,c_bus,control_signal[15],a_bus);
  alu #(32) ALU(clk,a_bus,b_bus,control_signal[21:16],N,Z,alu_output);
  shifter #(32) SHIFTER(alu_output,control_signal[23:22],c_bus);
  control_store CONTROL_STORE(clk,control_store_address_bus,control_signal);
  decoder DECODER(control_signal[3:0],decoder_output[15:0]);
endmodule
```

RAM

```
module ram #(parameter WORD_WIDTH = 8, ADDRESS_WIDTH = 32)
(clk,word_address,word_data,byte_address,byte_data,read,write,fetch);
  input clk,read,write,fetch;
  input [ADDRESS_WIDTH-1 : 0] word_address;
  input [ADDRESS_WIDTH-1 : 0] byte_address;
  output reg [WORD_WIDTH -1 : 0] byte_data;
  inout [(WORD_WIDTH*4)-1 : 0] word_data;

  parameter RAM_DEPTH = 128;

  reg [WORD_WIDTH-1:0] mem [0:RAM_DEPTH-1];
  reg [(WORD_WIDTH*4)-1 : 0] word_data_out;

  assign word_data = (read && !write) ? word_data_out : 32'bz;

  initial
```

```
  begin
    word_data_out <= 32'bZ;
    $readmemb("main_memory.list", mem); // memory_list is memory file
  end

  always @ (posedge clk)
  begin
    if (write && !read)
    begin
      mem[word_address] <= word_data[(WORD_WIDTH*1)-1 : (WORD_WIDTH*0)];
      mem[word_address+1] <= word_data[(WORD_WIDTH*2)-1 : (WORD_WIDTH*1)];
      mem[word_address+2] <= word_data[(WORD_WIDTH*3)-1 : (WORD_WIDTH*2)];
      mem[word_address+3] <= word_data[(WORD_WIDTH*4)-1 : (WORD_WIDTH*3)];
    end
  end

  always @ (posedge clk)
  begin
    if (!write && read)
    begin
      word_data_out[(WORD_WIDTH*1)-1 : (WORD_WIDTH*0)] <= mem[word_address];
      word_data_out[(WORD_WIDTH*2)-1 : (WORD_WIDTH*1)] <= mem[word_address+1];
      word_data_out[(WORD_WIDTH*3)-1 : (WORD_WIDTH*2)] <= mem[word_address+2];
      word_data_out[(WORD_WIDTH*4)-1 : (WORD_WIDTH*3)] <= mem[word_address+3];
    end
  end

  always @ (posedge clk)
  begin
    byte_data <= (fetch == 1) ? mem[byte_address] : 8'bz;
  end
endmodule
```

SHIFTER

```
module shifter #(parameter WORD_WIDTH = 8) (in,control,out);
  input [WORD_WIDTH-1 : 0] in;
  input [1:0] control;
  output reg [WORD_WIDTH-1:0] out;

  always @ (*)
  begin
    if(control==2'b01)
      out <= in>>>1;    //right_shift
    else if(control==2'b10)
      out <= in<<8;   //left_shift
    else
      out <= in;
  end
endmodule
```

TESTBENCH

```
module testbench(clk, reset);
  input clk,reset;

  wire [31:0] word_data;
```

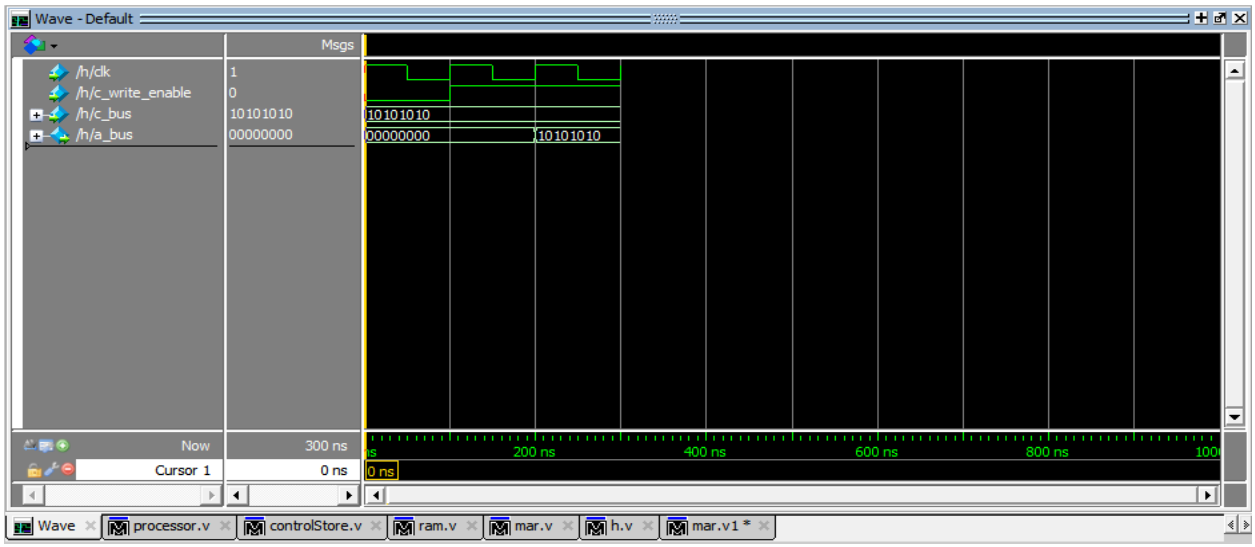
```
  wire [7:0] byte_data;
  wire [31:0] word_address;
  wire [31:0] byte_address;
  wire read,write,fetch;

  processor p(clk,reset,word_address,word_data,byte_address,byte_data,read,write,fetch);
  ram r(clk,word_address,word_data,byte_address,byte_data,read,write,fetch);
endmodule
```

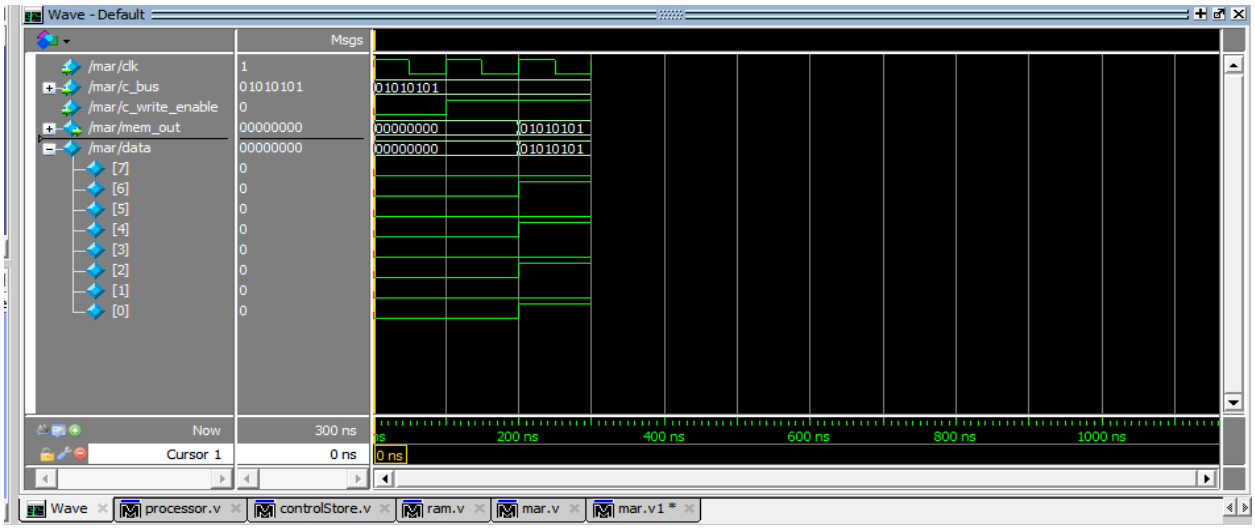
6. SCREENSHOTS

Simulation of module:

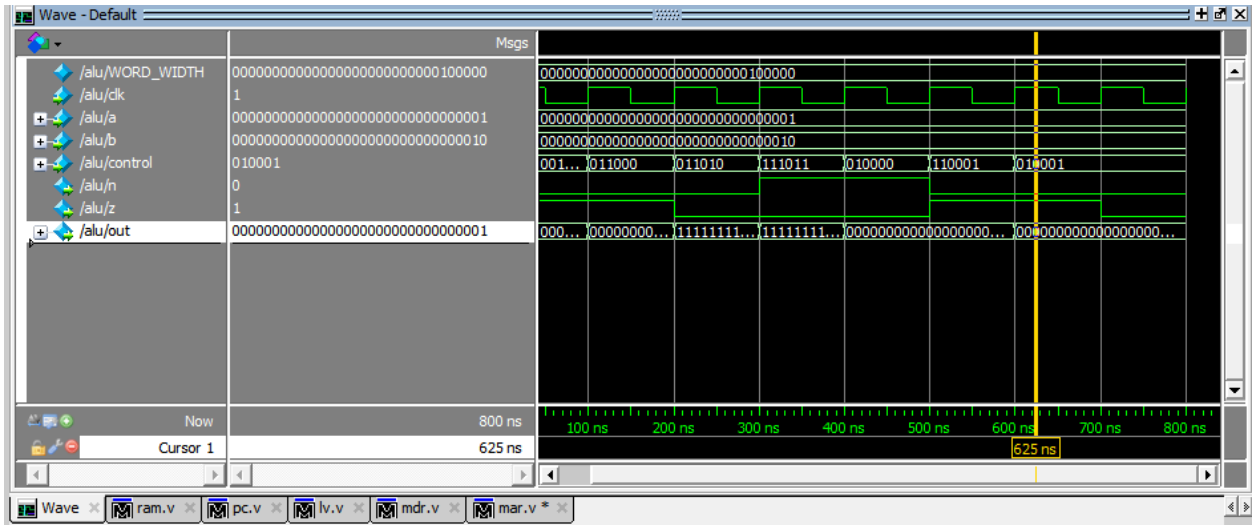
H



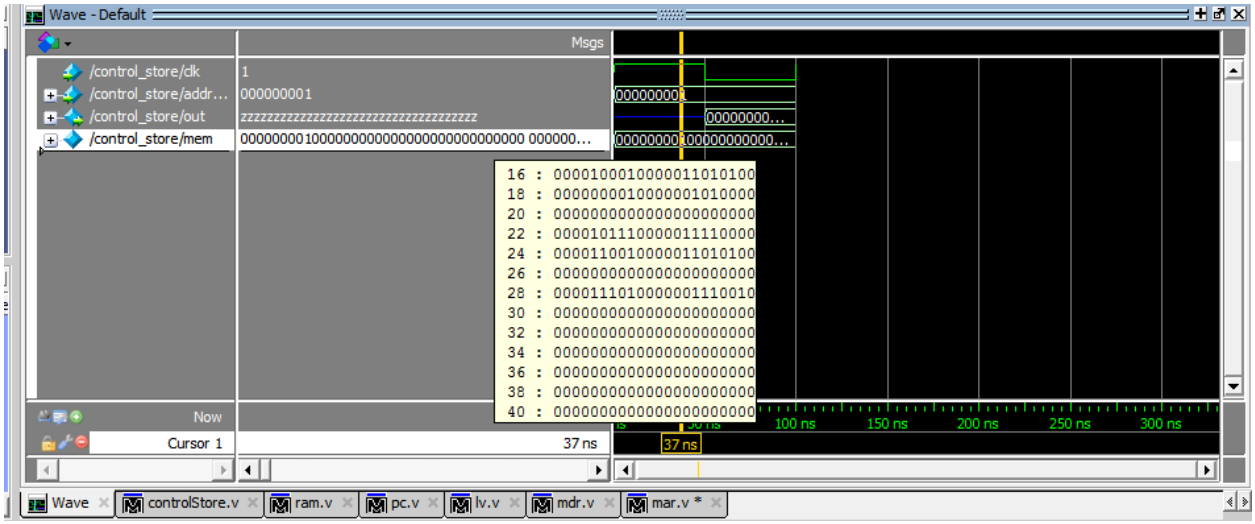
MAR



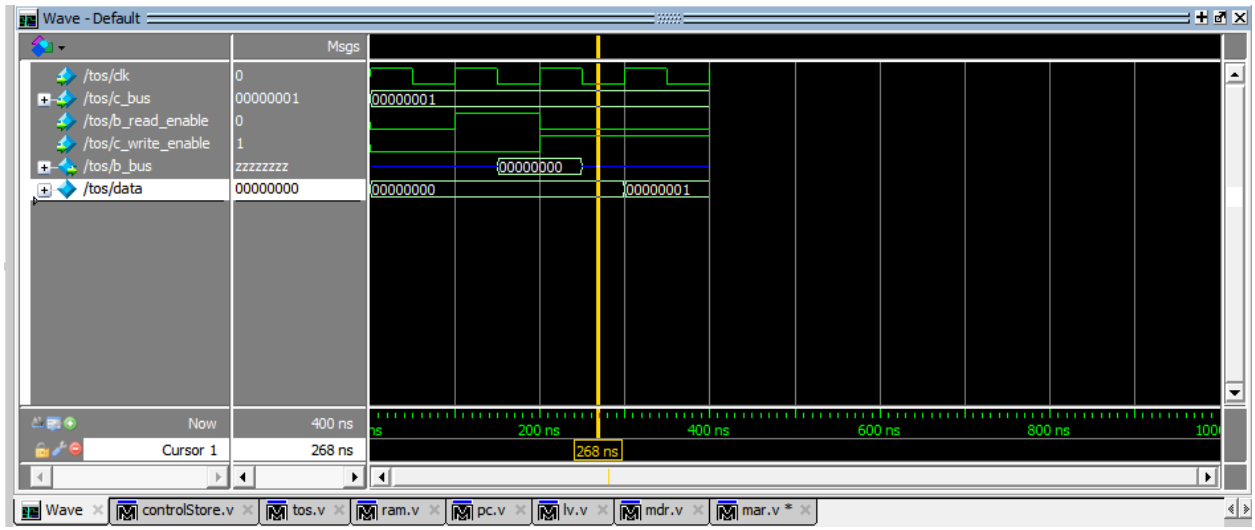
ALU



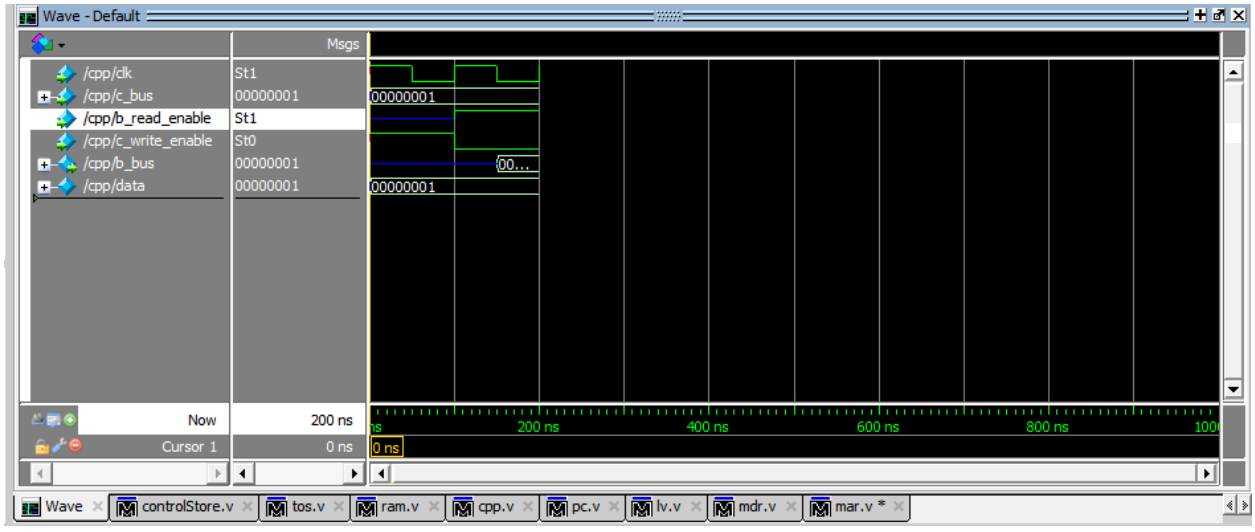
CONTROL STORE



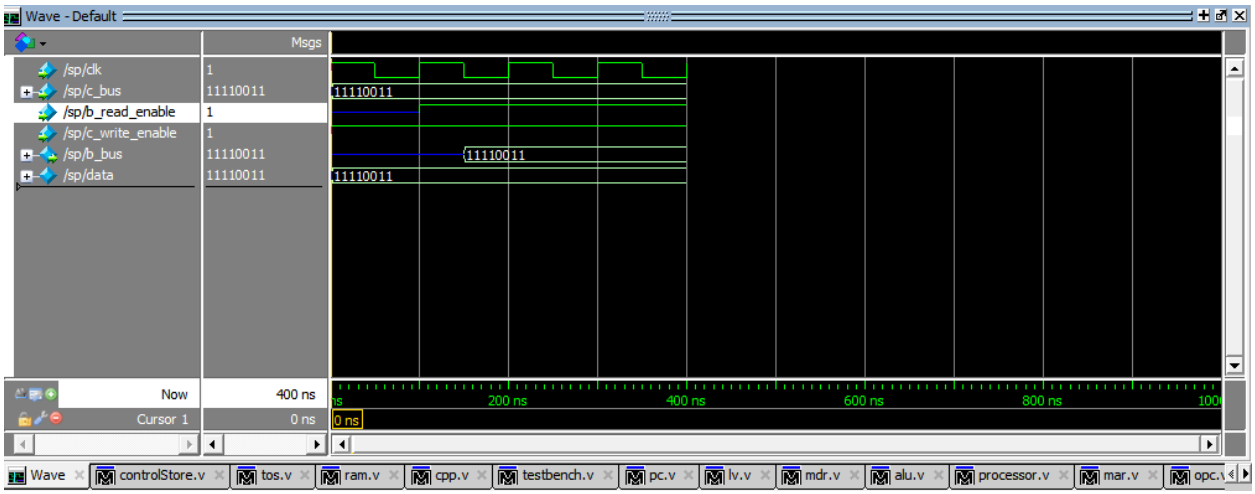
TOS



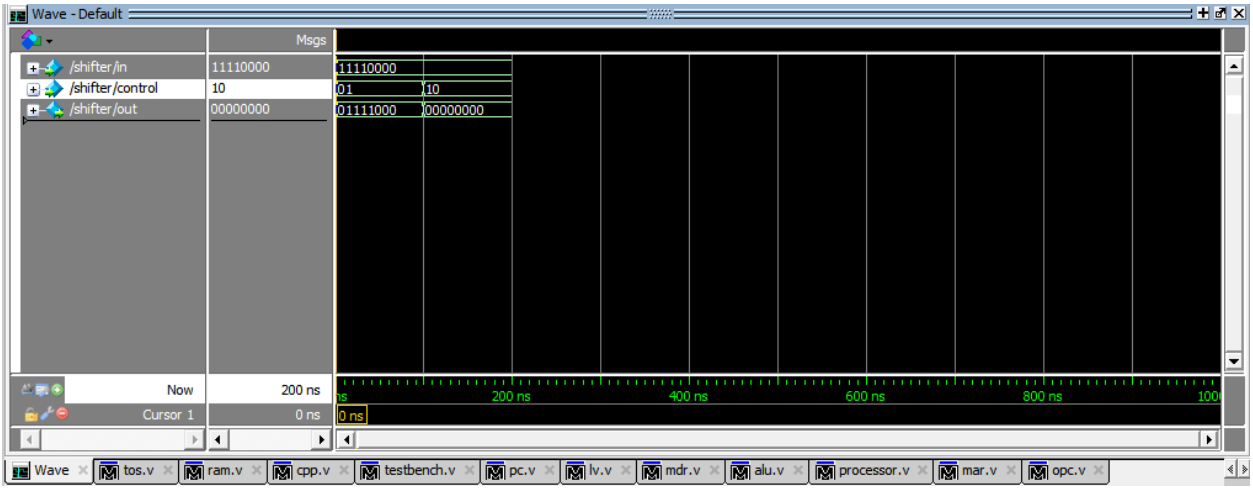
CPP



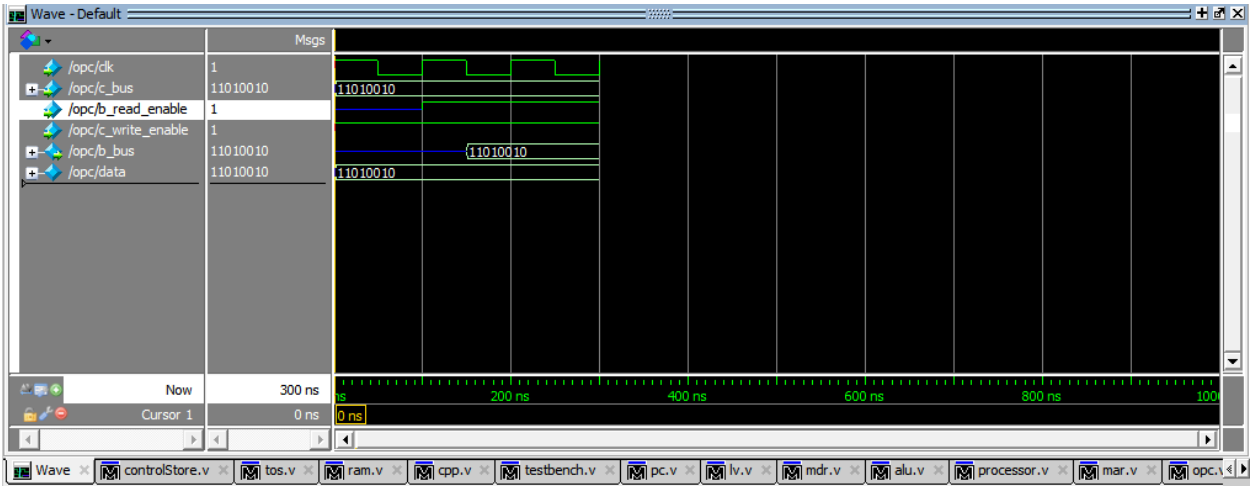
SP



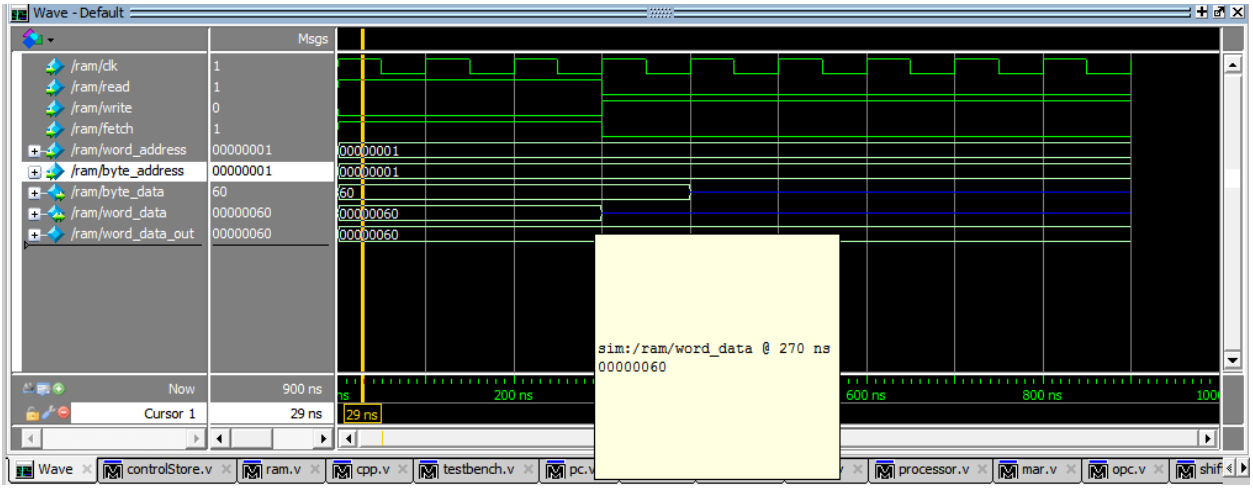
SHIFTER



OPC



RAM


Wave - Default
Msgs
/ram/clk
/ram/read
/ram/write
/ram/fetch
/ram/word_address
/ram/byte_address
/ram/byte_data
/ram/word_data
/ram/word_data_out
1
1
0
1
00000001
00000001
60
00000060
00000060
sim:/ram/word_data @ 270 ns
00000060
Now
900 ns
Cursor 1
29 ns
200 ns
600 ns
800 ns
Wave
controlStore.v
ram.v
cpp.v
testbench.v
processor.v
mar.v
opc.v

BIBLIOGRAPHY

1. Structured Computer Organization, Fourth Edition, Andrew S. Tanenbaum
2. Computer Architecture – A Quantitative Approach, John L. Hennessy and David A. Patterson
3. www.stanford.edu
4. Computer System Architecture, Third Edition, Morris Mano
5. vul.verilog.com
6. The Verilog Hardware Description Language by Philip R. Moorby, Donald E. Thomas, Fifth Edition