DIGITAL SIMULATION AND PROCESSOR DESIGN

DAVID CURRIE, ST JOHN'S COLLEGE

ENGINEERING AND
COMPUTER SCIENCE, PART II, 1998

SUPERVISOR – DR J SANDERS

Contents

1 Introduction

1.1 Background

Since the arrival of the third generation computer in the mid 1960s with the advent of the integrated circuit, manufacturers have been seeking to cram more and more transistors into smaller and smaller pieces of silicon. Designers have used this increased computing power to create ever more complex instruction sets in an attempt to bridge the semantic gap between high-level languages such as COBOL and Pascal and the low-level assembly language.

These so called Complex Instruction Set Computers (CISC), were encouraged by the slow speed of main memory compared to that of the CPU. Rather than call a slow library routine from memory to perform a common function, such as conversion from binary to decimal, it was tempting to simply add yet another instruction.

In the 1970s things began to change. Firstly, semiconductor RAM memory was no longer ten times slower than ROM. Writing, debugging and maintaining this complex microcode was also becoming increasingly difficult – a bug in the microcode means replacing all your installed ROMs. Thirdly, academics such as Tanenbaum were examining programs and discovering that, on average, over 80% of instructions were simple assignments, if statements and procedure calls. Of the assignments, 95% had one or no operator on the right hand side. Over 90% of all procedure calls had fewer than five parameters and 80% of the procedures had less than four local variables. The conclusion from this is simply that people do not write complex code.

A larger instruction set with more addressing modes requires a bigger and slower interpreter. The introduction of microprogramming had finally started to reduce the efficiency of computers. This heralded a new breed of Reduced Instruction Set Computers (RISC), the IBM 801 (1975), the Berkley RISC I (Patterson and Séquin, 1980) and the Stanford MIPS (Hennessy, 1984). These machines had the common goal of reducing the datapath cycle time. Some of the features that characterise RISC machines are given in Table 1.

Simple instructions taking one cycle
Only loads/stores reference memory
Highly pipelined
Instruction executed by the hardware
Fixed format instructions
Few instructions and modes
Complexity is in the compiler
Multiple register sets

Table 1- Characteristics of RISC machines

1.2 Overview of Project

The aim of this project was to design, build and test a digital circuit simulator to model processor circuits. This was to be used to compare and contrast the major architectural differences between the CISC and RISC architectures. The circuits were to simulate the lowest three levels shown in Figure 1, i.e. from the digital logic up to conventional machine level. The reason for this approach was to allow detailed analysis of the critical paths of various designs by calculation of delays across each component.

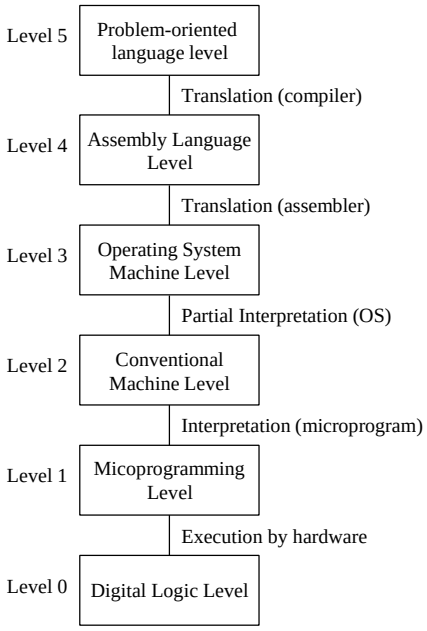

Figure 1 - Levels in a modern computer

1.3 Objectives

The objectives of the project were therefore defined as follows:

1. To design and build a digital circuit simulator and graphical interface
2. To design a circuit for, and then simulate, a CISC processor
3. To design a circuit for, and then simulate, a RISC processor
4. To compare the performance of these two processors using the simulations

Further background reading showed the division between RISC and CISC machines to be less than clear cut. Pipelining, for example, is not restricted to RISC machines. Microprogramming is sometimes present in RISC machines where the complexity of the control unit warrants its inclusion. Similarly, although RISC instructions may be fixed in length there are still usually three or more different formats. The final deciding factor in changing the objectives was that I wished to keep the circuits simple and this would be at odds with trying to demonstrate a complex instruction set.

The decision was therefore made to stick with a single instruction set, that used by the MIPS machines. This would be used to illustrate some of the ways in which the data path cycle time can be reduced, the key goal in the design of RISC machines.

2 Programming Language

An important initial decision was the choice of platform to write the program for, and in which language. Although PC/Windows was the preferred development platform, Sun/Solaris seemed a sensible option to allow the demonstration of the simulator in the Computing Laboratory.

It was important that the language should support the complex graphical user interface (GUI) that would be required for the input and simulation of circuits. Packages such as TCL/TK, Xlib, Xview and Motif would simplify this under X Windows. This would unfortunately restrict the development platform to Sun/Solaris. Similarly, the use of Visual C++, or similar development tool, would have meant the simulator could only run under Windows 95.

An object-oriented language seemed to be a reasonable choice. The logic gates that were to be the basis of the project fit easily with the idea of an object. Object methods correspond to the input and output of signals, and attributes to any internal state the gate may have.

A decision was taken to program the project in Java. The cross-platform portability of Java meant that the simulator could be developed, and demonstrated, on any supported platform including Solaris and Windows 95. Java is an object-oriented language and, although it was new to me, is a close relation to C++ with which I am well acquainted. The Java Advanced Window Toolkit (AWT) classes make producing windows, menus and simple graphics easy. Version 1.1 was used which includes a new event-based model that proved to be particularly useful. The Java Foundation Classes (JFC), nicknamed Swing, were also used (although the first full release was not until near the end of the project). These provide additional GUI components such as toolbars, popup menus and scroll panes with helpful features such as automatic double buffering. The Swing classes are to be included as standard in Java version 1.2.

3 Simulation

3.1 Strategies

Before designing the simulator it was first necessary to decide on the strategy to be used. Three possible options presented themselves and these are discussed below.

3.1.1 Asynchronous Simulation

This strategy at first appears to be the simplest. When we change the input of a logic gate the output is also changed. Unfortunately, this is an oversimplification. Although the project is to simulate sequential circuits, signals still travel in parallel.

Consider the simple circuit shown in Figure 2 whose output should always be zero. Let the input be initially zero. Now let the input rise to one. Do we set the top input to one first or the bottom? If we set the bottom first then the output will remain zero, but if we set the top first then the output will temporarily go to one before we set the bottom.

Figure 2- Parallel signals

How about the circuit shown in Figure 3? If before the feedback loop is closed the top input is one and the bottom input is zero, the output would be one. Closing the loop now sets the top input to one. This sets the output to one which, in turn, sets the top input to one. This infinite loop could be prevented by only setting the output when the inputs change value but this is an additional complication.

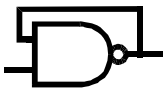

Figure 3 - Oscillation

Now take the circuit of Figure 3 with both inputs initially one and the output zero. When the loop is closed it will set the top input to zero. This sets the output to one, which in turn sets the input back to one. This time even more care needs to be taken to ensure that, while simulating the infinite oscillation, we do not prevent the rest of the circuit from operating.

3.1.2 Synchronous Simulation

A synchronous simulation strategy depends on a clock, quite separate from any that may exist in the circuit being simulated. The clock consists of three phases. In the first phase all the logic gates in the circuit read their inputs, in the second they calculate the new output values and in the third these are written to the outputs. This separation means that the signals can be sent down parallel wires in any order during the output phase as they will not affect the inputs of other gates until the next clock cycle.

The oscillations in the circuit of Figure 3 will now take place but governed by the rate of the clock. This scheme introduces the idea of a propagation delay across gates. For example, in the circuit of Figure 4, when the input changes from a zero to a one it is only a clock cycle later when the output is written that the change in output becomes visible.

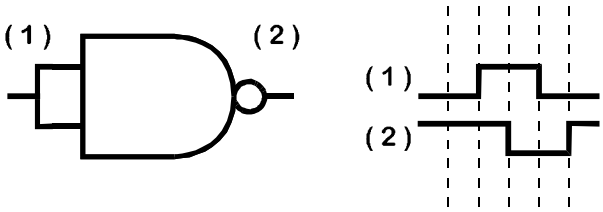

Figure 4 – Propagation delay across logic gate

3.1.3 Semi-Synchronous

This strategy is based on the synchronous strategy but allows some variation in the length of the clock cycles. If there is not time to update all of the gates within the length of one clock cycle it is extended to allow this to take place. This prevents problems occurring when some gates are reading their next input before other gates have finished writing their outputs.

3.2 Implementation

It was decided to use the semi-synchronous simulation strategy. An abstract Java class called Package was created to represent electronic packages. The methods addInput() and addOutput() can be used to respectively add inputs and outputs to the package and are used in the package's construct() method. The initialise() and simulate() methods must be defined when Package is extended. The former is called when the simulation is reset and can therefore be used for initialising state. The latter is called on each clock cycle between reading the inputs and writing the outputs. These two methods may make use of the getInput() and setOutput() methods which, as there names suggest can be used to retrieve the current value of an input or set the value of an output.

The example below illustrates the simplicity of this approach in the design of a NAND gate. In the construct() method two inputs 'a' and 'b', and an output 'c', are added to the package. As well as the name of the connections a number is used to specify their width in bits. As the NAND gate has no state, the initialise() method is empty. The simulate() method gets the value of the two input connections and sets the output connection. Here, the methods getInput() and setOutput() specify the index of the bit whose boolean value is required. If required these methods can be used without the index, in which case the whole input is returned, or output set, using a Binary object. The Binary class has methods for performing binary operations such as add, subtract and negate to make the process of writing the simulation classes easier.

```
class NANDGate extends Package {

    public void construct () {
        addInput("a", 1);
        addInput("b", 1);
        addOutput("c", 1);
    }

    public void initialise() {
    }

    public void simulate() {
        setOutput("c", 0, !(getInput("a", 0) && getInput("b", 0)));
    }

}
```

The Package class shields the user from the intricacies of the simulation strategy. When a simulation is started a separate Thread is created which provides the clock. At the beginning

of each clock cycle the readInputs() method is called on all of the packages contained within the simulation. This method stores the current levels of the inputs within the package, shielding it from any changes which may take place during the current clock cycle. The calls to getInput() return the stored levels. The clock then calls simulate() on all the components. Calls to setOutput() simply store the new outputs within the package. Only when the clock calls writeOutputs() are the values written to the output connections. The reason for this will become clear later.

When the value of an output is set, if there any wires leaving the connection their values are in turn set. These wires then check if they have an input connection at the other end and, if so, set its value. Although more than one wire may leave a particular bit of an output connection, only one wire may arrive at each bit of an input connection to prevent conflict over boolean levels.

The levels used are the Java keywords true and false corresponding to the binary values one and zero respectively. It was decided to simplify the simulation by not providing separate levels such as floating low/high. All inputs and outputs are zero unless set otherwise.

4 The Editor and Simulator

This section outlines some of the features associated with the editing and simulation package, hopefully giving an idea of both the ease of use and the way in which they function.

On starting up ,the user is presented with a blank grid as shown in Figure 5. To this they may add an input, output or package. Packages may be selected from the compiled Java class files saved on disc. As a starting point class files were written for a NAND gate, Clock, one level source and zero level source.

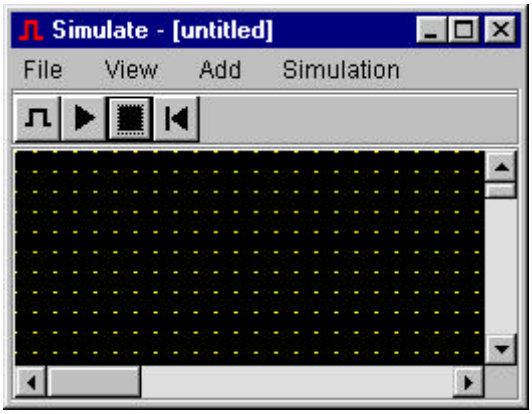

Figure 5 – Screen shot on startup

The connections of packages can then be joined using wires. As connections may have widths of more than one bit the user has to state which bits the wire is to be joined to. Appropriate checks are made to ensure that wires do not overlap when inputting to a connection. Packages and wires can be repositioned simply by dragging them or deleted using a popup menu. Resting the mouse over a package or connection brings up a text box with the component name as in Figure 6.

To aid clarity packages may not overlap. The decision was made that wires should always start at a package connection due to the complications associated with implementing wire joints. Wires are therefore allowed to be placed on top of each other as they may well be carrying the same signal.

If the circuit is itself just a package then appropriate outputs and inputs can be added and named. Saving the circuit results in a sim file that can be loaded into another circuit using the ‘add package’ option. This allows a modular approach to design. The simple circuit shown in Figure 6 shows how a Clock.class package and a NOTGate.sim package have been connected together. When this circuit is saved it does not record the definitions of the included packages. Instead, either the name of the class or pathname of the sim file is recorded. This means that if the inner workings of either of the packages are altered the new versions will be used next time the circuit is loaded. The input and output connections must obviously not be changed.

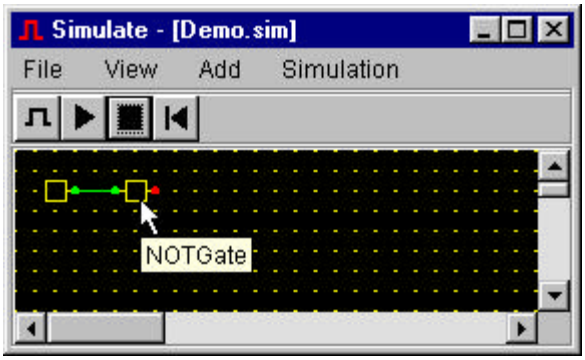

Figure 6 - Clock attached to a NOT gate

The buttons along the top of the window duplicate those on the Simulation menu. These allow a simulation to be stepped, run, stopped or reset. The priority of the thread that performs the simulation was kept low to allow these to interrupt the current action. During simulation, connections and wires which have one or more bits set appear in green, else they appear red. This is a particularly useful feature when watching the control lines of processor circuits.

Three other methods exist to allow the user to view the simulation process in more detail. A probe can be attached to a connection producing a new window displaying the output traces in real time. Figure 7 shows probes connected to the input and output connections of the NOT gate of Figure 6. It is possible to zoom in on the trace and here the delay across the gate can clearly be seen. (Note that time increases from right to left.) More useful in the case of multi-bit connections are monitors that display the binary value of the monitored connection.

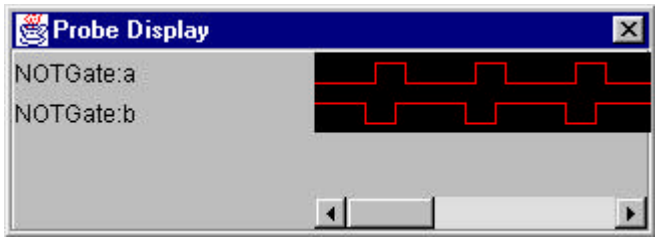

Figure 7 – Probe display for circuit of Figure 6

The third method allows the user to examine the state of packages. Using a Java technique called reflection the program checks to see if any of a class's attributes have been defined as public. If so, the attribute's name and current value are displayed. This is particularly useful for viewing the contents of simulated memory.

5 Basic Components

This section shows how all the combinatorial logic required for a processor circuit can be built up from a complete logic gate such as the NAND gate.

5.1 NAND Gate

The NAND gate was chosen as the basis for all the circuits used in this project. The Java source code for this package has already been given above.

5.2 Logic gates

Other basic logic gates can easily be built up using just NAND gates. The NOT gate is simply implemented by connecting together the two inputs of the NAND gate as shown in Figure 8. The delay given in brackets is the maximum time it will take a change in one of the inputs to the circuit to be fully represented in the output of the circuit normalised by the delay across a NAND gate.

Figure 8 – NOT gate implementation (Delay=1)

An AND gate can therefore be implemented by placing a NOT gate after a NAND gate as shown in Figure 9.

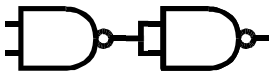

Figure 9 – AND gate implementation (Delay=2)

The number of gates increases once more when it comes to implementing an OR gate as shown Figure 10.

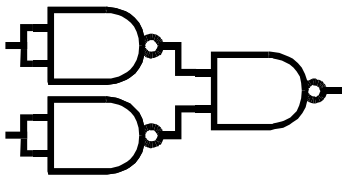

Figure 10 – OR gate implementation (Delay=2)

The complexity increases yet again to implement an XOR gate as shown in Figure 11. Note that this circuit includes a ‘hazard’ as some signals only have to pass through two gates and others through three. A change in input may therefore result in the output changing after a delay of two but will only settle to its true value after a delay of three.

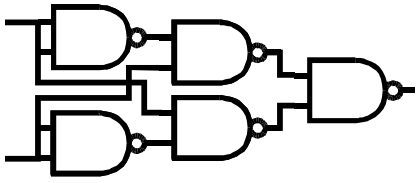

Figure 11 – XOR gate implementation (Delay=3)

5.3 Multiplexor

A simple multiplexor, which takes two one-bit inputs, and a select line to choose between them, is illustrated in Figure 12. The corresponding circuit symbol is shown in Figure 13. More generally to select between 2^n m-bit inputs will require a circuit with delay five and $(2^n(1 + m) + n + 1)$ gates to implement the decoder and selection logic.

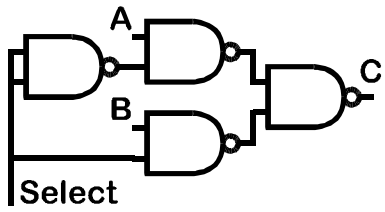

Figure 12 - Two to one one-bit multiplexor implementation (Delay=3)

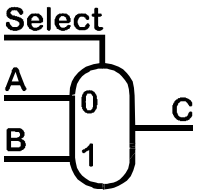

Figure 13 - Two to one one-bit multiplexor circuit symbol

5.4 Arithmetic Logic Unit (ALU)

An ALU can be built up in sections. First we consider a half adder used to add two one-bit numbers together. The truth table required can be seen in Table 2. The carry and sum terms are, respectively, the AND and XOR of the two inputs. The half adder can therefore be simply implemented using the circuit of Figure 14.

A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 2 - Half adder truth table

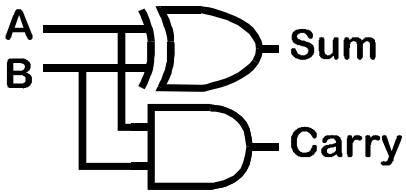

Figure 14 – Half adder implementation (Delay=3)

When we are adding numbers of more than one bit we also have to consider the carry-in from the summation of the lower bits as shown in Table 3. The expressions for carry-out and sum seem complicated but can be implement easily using two half adders as shown in Figure 15.

A	B	Carry In	Carry Out	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 3 - Full adder truth table

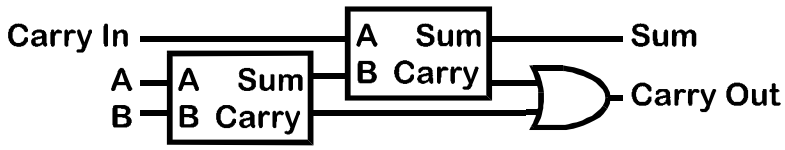

Figure 15 – Full adder implementation (Delay=6)

The ALU we are building is required to do more than just add. It must also subtract, AND, OR and compare numbers. The one bit ALU of Figure 16 can perform all of these operations as will become apparent when they are joined together as in Figure 17. The rectangle marked with the plus sign is the full adder of Figure 15. Notice also the use of a multiplexor to select the required result.

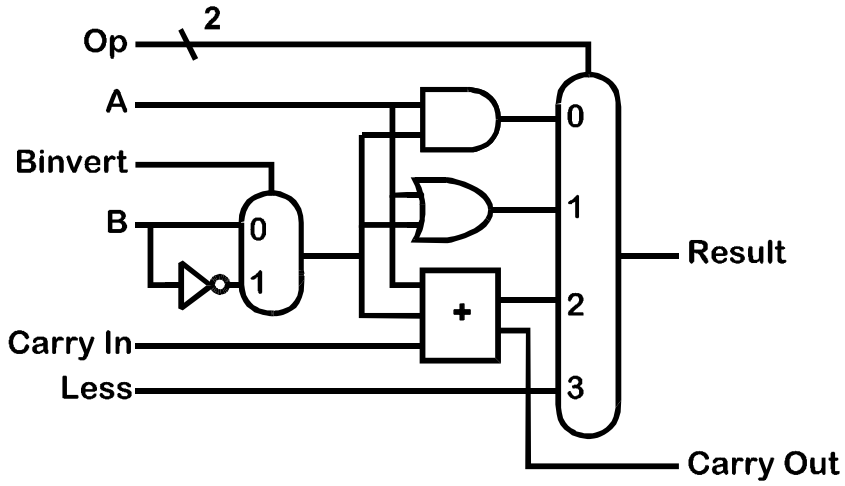

Figure 16 – One bit ALU implementation (Delay=15)

Figure 17 shows how these one-bit ALUs can be connected to create a 32-bit ripple carry ALU. The operation control line is used to select the output from either the AND gate, the OR gate, the adder or the less input. Subtraction is done by negating the B input. Negation in two's complement involves inverting all the bits and adding one. This is done in the ALU by setting the Binvert line on all the one-bit ALUs and setting CarryIn for ALU0 to one.

The less input of the one-bit ALU is used for the set-on-less-than instruction. The last one-bit ALU has an additional output, set, which takes the output of the last adder. If subtracting B from A gives a negative answer (i.e. A < B) then the output from the last adder will be one. Therefore, the output of the complete unit will be one if A < B, else it will be zero.

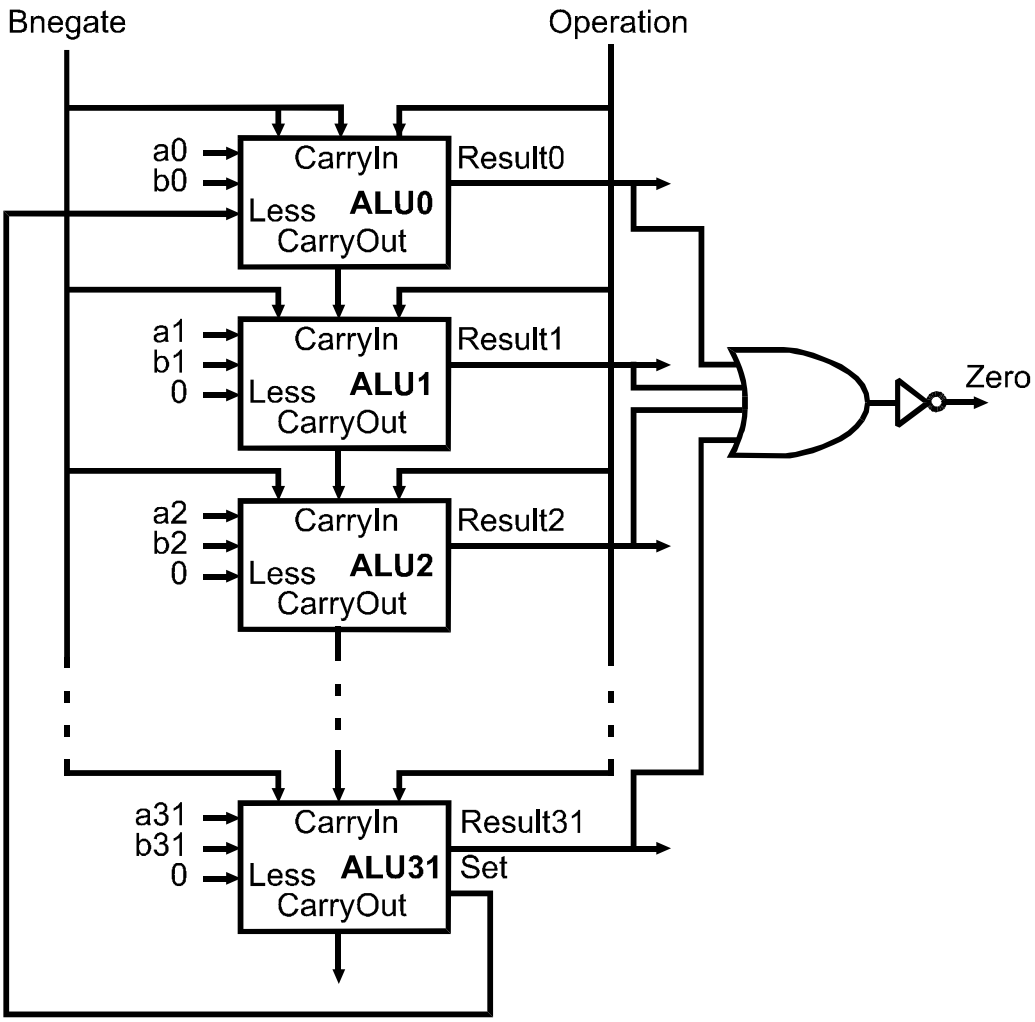

Figure 17 – 32 bit ALU implementation (Delay=173)

The ALU symbol used in later circuits is shown in Figure 18.

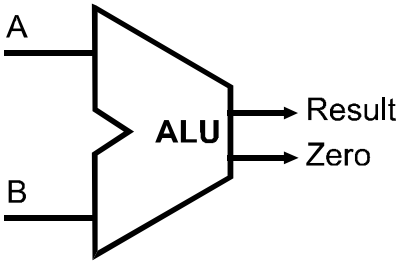

Figure 18 – ALU symbol

Note the delay across this design of ALU is very large. This is because when performing an operation using the full adders we must wait for the carry bits to 'ripple' through from ALU0 to ALU31. The signal b0 takes a maximum delay of four to reach the input of the first adder. The delay from carry-in to carry-out is five and from carry-in to sum is six. Therefore the last sum is finished after a delay of 4 + (31*5) + 6 = 165. The delay across the output multiplexor is five and there is a further delay of three to calculate zero. Therefore, the total delay across the ALU from start to finish can be up to 173. The speed can be increased using carry-lookahead but we shall not consider this process here. With 38 NAND gates required to implement a one-bit ALU over 1200 are needed for the 32-bit version.

5.5 Register

Possibly one of the most important components in processor design is the register, used to store data. Again, we will build in a modular fashion. First, we start with the D-type latch. The circuit shown in Figure 19 takes data at input D and when the clock signal, CK, is high transfers it to the output Q. When the clock goes low the output Q will hold the last value it had until the clock goes high again.

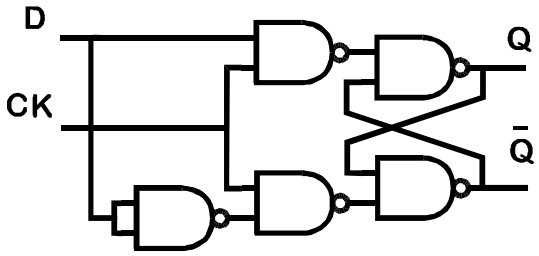

Figure 19 – D-type latch implementation (Delay=3)

This is not entirely satisfactory as while the clock is high the output Q follows the input D. What we really require is that the data is transferred from input to output on either the negative or the positive going edge of the clock. One method of achieving this is to add edge-triggering circuitry to the clock. For example, taking the AND of the clock signal and it's inverse will result in a peak at the positive going edge of the clock due to the delay through the inverter.

However, the design used in this project is the master slave flip-flop shown in Figure 20. When the clock is high the output of the first latch follows the input. The negated clock signal means that the output of the second latch remains constant. When the clock goes low, the output of the first latch remains constant and the output of the second follows it, i.e. the input at that moment is transferred to the output.

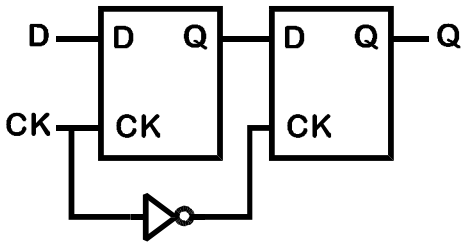

Figure 20 – Flip-flop implementation (Delay=4)

Now all that remains to create an n-bit register is to put n flip-flops in parallel as shown in Figure 21. Although the flip-flop requires eleven gates the clock inverter can be shared and therefore a 32-bit Register would require (32x10)+1 = 321 gates.

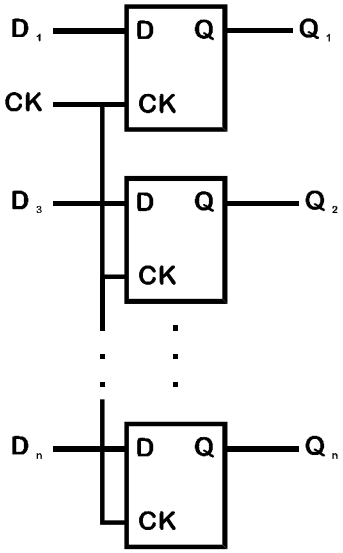

Figure 21 – Register implementation (Delay=4)

5.6 Register file

Often an array of registers is required for storing information. The register file shown in Figure 22 has 32 registers. In a single clock cycle it is possible to write to one register and read from two. Of course the outcome cannot be guaranteed if an attempt is made to read a register that is also being written to. The quoted delay is between the clock changing and the data being available for reading. By convention Register0 is wired to zero. Reads from this register return zero and writes to the register have no effect.

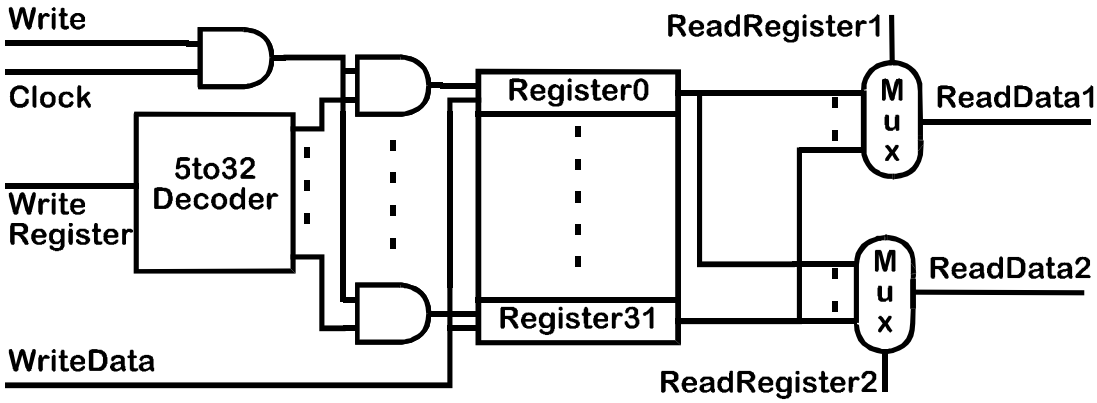

Figure 22 – Register file (Delay=11)

5.7 Memory

Although this project is dealing with processor design, in order for the processors to be of any practical use they must be connected to data and instruction memory. There are two common types of memory, SRAM and DRAM. We will not deal with the complexities of memory here but in line with the other elements of our computer architectures we require a delay time for read/write access to the memory. Memory and the associated bus transfers to and from memory are slow and to reflect this we shall give memory a delay of 180, similar in magnitude to that of the ALU.

5.8 Component Abstraction

The speed when simulating some of the components above was prohibitively slow due to the large numbers of gates present in the designs. It was therefore decided to increase the level of abstraction by creating Java class files equivalent to the sim files. This is where the buffering on the output connections of a package is used. The addOutput() method can take an additional parameter specifying the delay that is added to any output signal from that connection.

By using this abstraction we lose a certain amount of the detail from the simulations. For example, although we are simulating the propagation delay, race conditions will no longer appear in the outputs. The code below shows how register files are implemented in this fashion.

```
class RegisterFile extends Package {

    private boolean previousClock;
    public Binary registers[];

    public void construct() {
        addInput("RegWrite", 1);
        addInput("ReadRegister1", 5);
        addInput("ReadRegister2", 5);
        addInput("WriteRegister", 5);
        addInput("WriteData", 32);
        addInput("Clock", 1);
        // Note delay of 11 on outputs
        addOutput("ReadData1", 32, 11);
        addOutput("ReadData2", 32, 11);
    }

    // Initialise all registers to be empty
    public void initialise() {
        registers = new Binary[32];
        for (int i = 0; i < 32; i++) {
            registers[i] = new Binary(32);
        }
    }

    public void simulate() {
        // Check for negative going edge
        if (previousClock && !getInput("Clock", 0)) {
            // Set outputs to values of requested registers
            setOutput("ReadData1",
                registers[getInput("ReadRegister1").intValue()]);
            setOutput("ReadData2",
                registers[getInput("ReadRegister2").intValue()]);
            // Check if RegWrite high and not writing to Register0
            if (getInput("RegWrite", 0) &&
                    (getInput("WriteRegister").intValue()!=0)) {
                // Write input to register
                registers[getInput("WriteRegister").intValue()] =
                    getInput("WriteData");
            }
        }
        previousClock = getInput("Clock", 0);
    }

}
```

6 Simplified MIPS instruction set

The instruction set around which the following processor is designed is a subset of that for the MIPS architecture. The first RISC MIPS chip was designed and fabricated by John Hennessy at Stanford in 1981. The instructions that we shall implement are outlined in Table 4. These are sufficient to show some of the design problems and to implement a simple program.

Instruction	Example	Meaning
add	add $1, $2, $3	$1 = $2 + $3
subtract	sub $1, $2, $3	$1 = $2 - $3
and	and $1, $2, $3	$1 = $2 & $3
or	or $1, $2, $3	$1 = $2 | $3
load word	lw $1, 100 ($2)	$1 = Memory[$2+100]
store word	sw $1, 100 ($2)	Memory[$2+100] = $1
branch on equal	beq $1, $2, 100	if ($1==$2) go to PC+4+(4×100)
set on less than	slt $1, $2, $3	if ($2 < $3) $1 = 1 else $1 = 0
jump	j 10000	go to 10000

Table 4 – Simpified MIPS instruction set

The numbers preceded by dollar signs indicate registers in a register file. This is a key feature of RISC architectures. As access to memory is slow, computation is speeded up by storing numbers within the processor in the register file. Load and store are the only two instructions that access memory, all others carrying out operations between registers. This also leads to a simplified instruction format. All three of the 32-bit MIPS formats are supported by the designs in this project.

7 Single-Cycle Processor

We shall start by designing a simple datapath through combining those required for each of the three instruction formats. This will take a single clock cycle per instruction.

7.1 Instruction Fetch

Common to all the instruction formats is the instruction fetch stage. To simplify matters we use separate memory for instructions and data. A 32-bit register called the program counter (PC) holds the address in bytes of the next instruction in memory to be executed. The register ensures that the address stays constant while the instruction is read from memory. As each of the instructions occupies four bytes we increment the program counter by this much each clock cycle to reach the next instruction. Figure 23 shows the simple circuitry required to perform the operation.

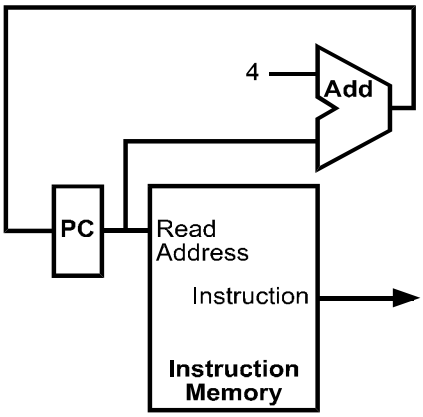

Figure 23 – Instruction Fetch

7.2 R-format Instruction

The next stage of the cycle involves decoding the instruction and setting up the required datapath. The R-format instruction is used for the arithmetic and logical operations. Table 5 shows the fields contained within the instruction. The opcode is the same for all R-format instructions.

Field	Bits	Meaning
op	6	Opcode (000000)
rs	5	First source register
rt	5	Second source register
rd	5	Destination register
shamt	5	Shift amount
funct	6	Arithmetic/Logic function

Table 5 - R-format instruction fields

Table 6 gives an example instruction, add $1, $2, $3. Each of the arithmetic/logic operations has a function code associated with it. Due to the fixed format of the instruction it is easy to retrieve any required field.

op	rs	rt	rd	shamt	funct
000000	00010	00011	00001	00000	100000

Table 6 – Instruction for add $1, $2, $3

The R-format instructions all have in common that they take data from the two source registers, perform an ALU operation on that data and then write the data back to a third register. The datapath required is therefore that shown in Figure 24.

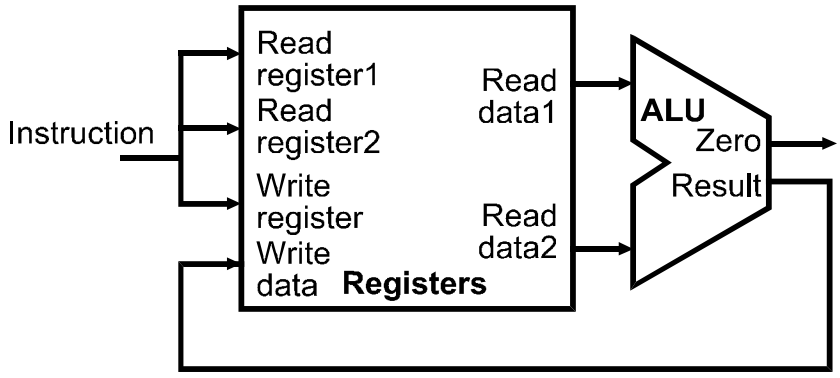

Figure 24 – R-format datapath

7.3 I-format Instruction

The I-format instruction is used for instructions that use immediate address, e.g. data transfer and branching. The format of the instruction is given in Table 7. Here the opcode alone is used to identify the particular operation that is required.

Field	Bits	Meaning
op	6	Opcode
rs	5	First register
rt	5	Second register
address/immediate	16	Address or immediate value

Table 7 – I-format instruction fields

For the load word and store word operations, the last field contains the offset of the required memory location from the address given in the first register. This facilitates indexed addressing. The data is either stored from or loaded to the second register. An example instruction is shown in Table 8.

op	rs	rt	address/immediate
100011	00010	00001	0000000001100100

Table 8 - Instruction for lw $1, 100($2)

Both instructions require us to read the address from register rs and add the offset. The ALU can be used to perform this addition provided the offset is sign extended to 32 bits. For a store operation we want to read the data in register rt and write it to this address, whereas, for a load operation we want to read the data at this address and write it into register rt. The datapath required is shown in Figure 25. It is up to the control logic to assert the appropriate read and write signals on the register file and data memory.

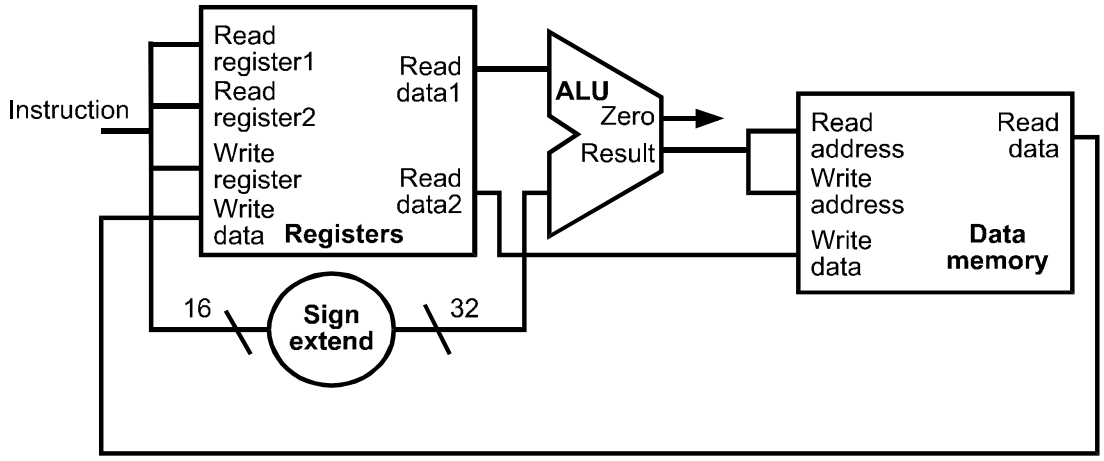

Figure 25 – Load/Store datapath

The second type of I-format instruction we need to consider is the branch instruction as illustrated in Table 9. Here the two registers are those whose contents we wish to compare and, if they are the same, we branch forward the number of instructions given by the offset.

op	rs	rt	address/immediate
000100	00001	00010	0000000001100100

Table 9 – Instruction for beq $1, $2, 100

The required datapath for the beq instruction is therefore that shown in Figure 26. The data is read from the two registers and the ALU performs a subtraction operation. The Zero output of the ALU can therefore be used by the branch control logic to determine if the branch should take place. The target address for the branch is calculated by sign extending the address/immediate field and then shifting the result two to the left to get the offset in bytes. This is then added to the value of (PC+4).

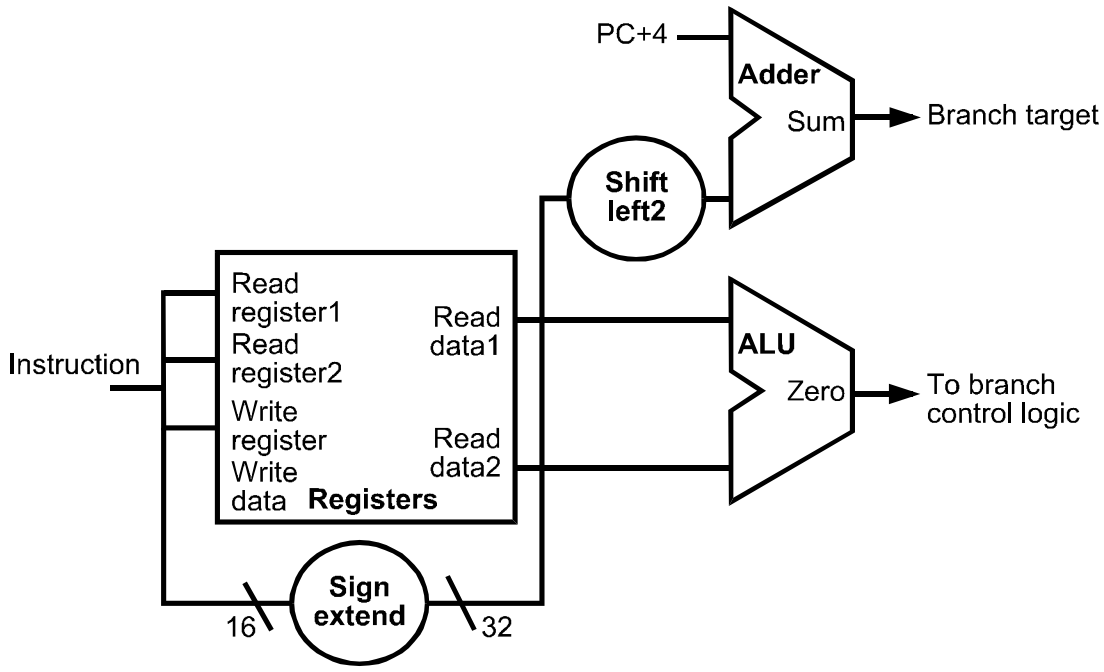

Figure 26 – Branch datapath

7.4 J-format Instruction

The third instruction format we shall consider is used solely for jump operations. The two fields and an example instruction are shown in Table 10 and Table 11 respectively.

Field	Bits	Meaning
op	6	Opcode (000010)
target address	26	The target address for the jump.

Table 10 – J-format instruction fields

op	target address
000010	00000000000000100111000100

Table 11 – Instruction for j 10000

The datapath for this instruction is trivial. If we wish a jump to take place all that we require is that the PC is set to the required target address. Once again the address given here is in bytes but, even after shifting, this still only gives us 28 bits. The remaining four are taken off the top of (PC+4).

7.5 Combining the Datapaths

All that is required now is to combine the datapaths of the three types of instruction. This is done using multiplexors and control logic. The main control unit shown in Figure 27 takes the opcode part of the instruction and decodes it. It then sets up the required datapath for the given instruction using the select lines for the multiplexors and the read/write lines of the register file and memory. Table 12 shows the required settings of the ten control lines for each instruction type. In this simple design, where we only need to distinguish between the five opcodes, the control unit can be implemented using either discrete logic gates or a small programmable logic array (PLA).

Instruction	Reg Dst	ALU Src	Mem toReg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op
R-format	1	0	0	1	0	0	0	0	10
lw	0	1	1	1	1	0	0	0	00
sw	X	0	X	0	0	1	0	0	00
beq	X	0	X	0	0	0	1	0	01
j	X	X	X	0	0	0	0	1	XX

Table 12- Control lines

The ALU control creates the Bnegate and Operation signals required by the ALU. How it does this depends on the signal it receives from the control unit as shown in Table 13.

ALUOp	Operation required
00	Add
01	Subtract
10	Use function code

Table 13- ALUOp control line decoding

A branch is required if the branch control line is set and the output of the ALU is zero. This test is implemented simply by an AND gate.

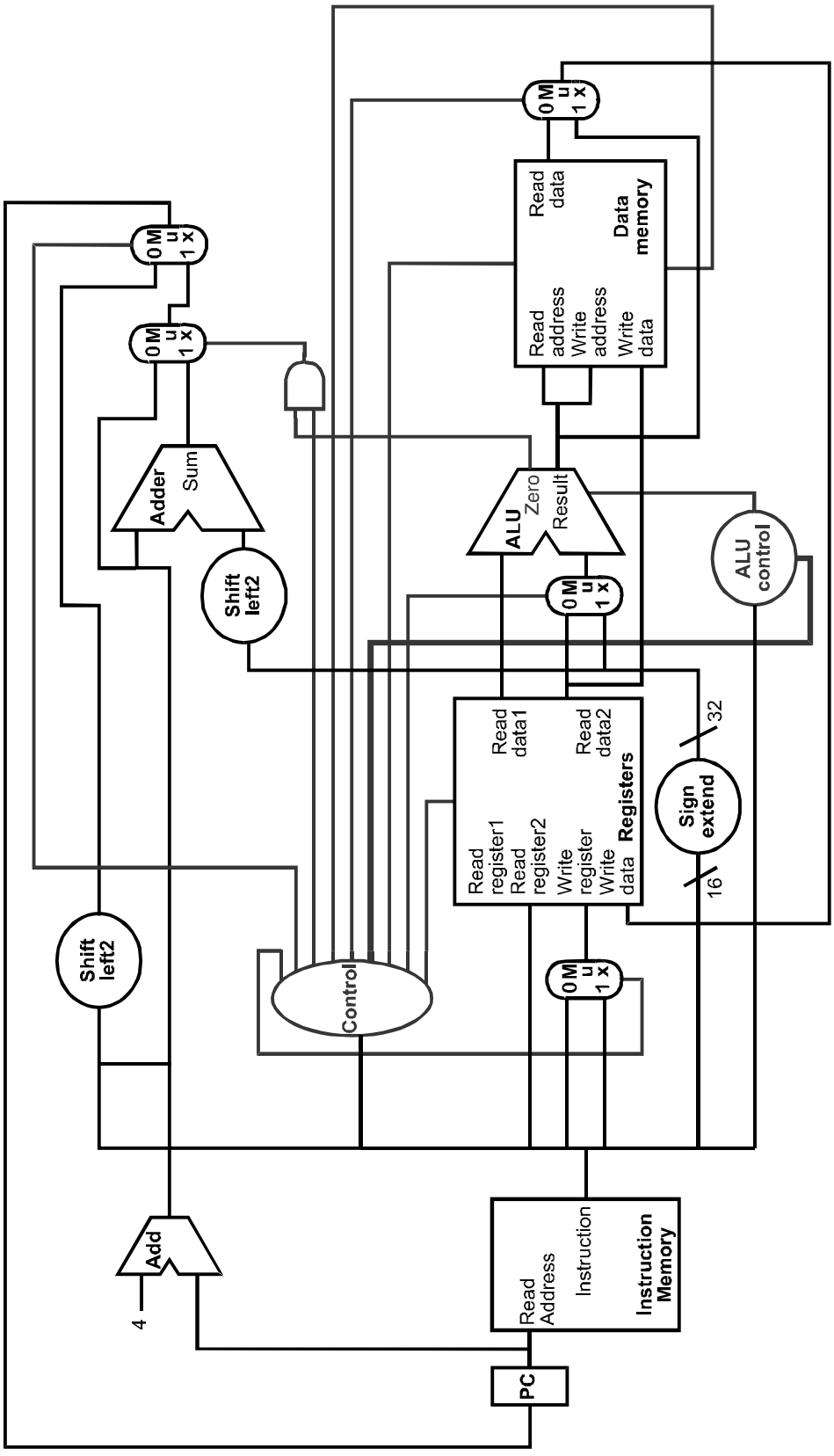

Figure 27 – Single-cycle processor

7.6 Simulation

The circuit is now ready to be entered into the simulator as shown in Figure 28. (The annotations have been added as the reader cannot view the package names as a user would.) As with the basic components, specialised components such as the control unit were first built

up from NAND gate level in the editor and then, after testing, replaced by a Java class file. The only addition to the circuit given in Figure 27 is that of the clock (bottom left). It has been broken down in five sub-phases to ensure that the data passes correctly from left to right through the circuit.

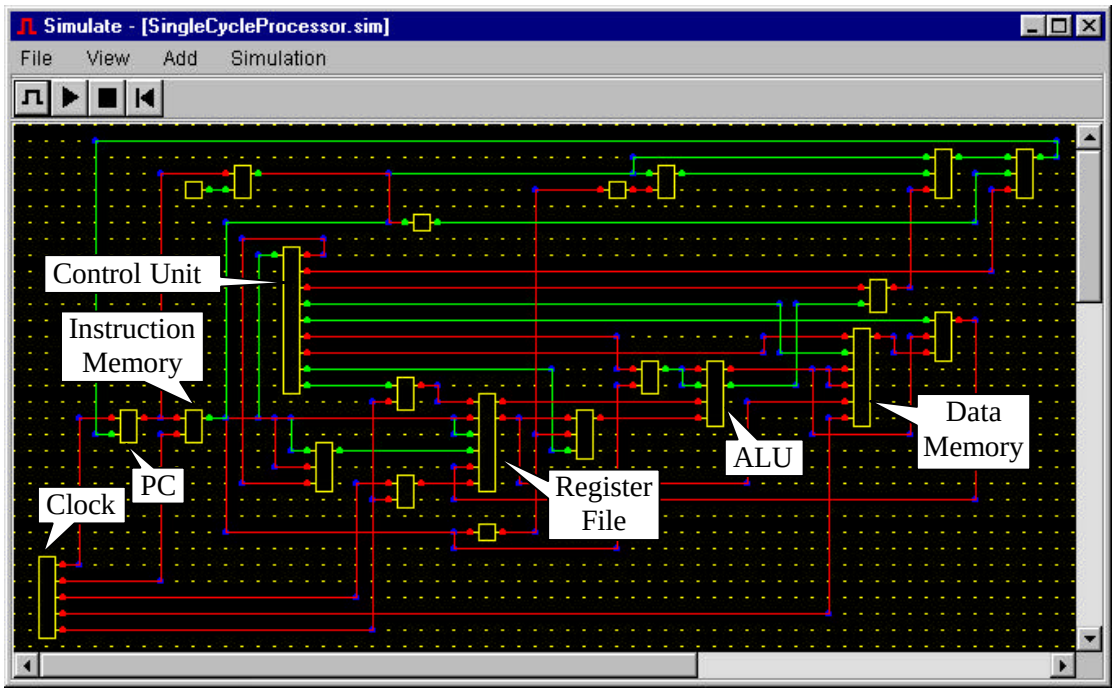

Figure 28 – Single-cycle processor simulation

The simulation was tested with the program shown in Table 14 which takes the numbers from memory locations 4 and 8, multiplies them by repeated addition and then stores the answer in memory location 12. As we have not implemented any instructions for adding or subtracting an immediate value, memory location 0 contains the value 1 used by the program.

```
lw $1, 0 ($0)
lw $2, 4 ($0)
lw $3, 8 ($0)
add $4, $2, $4
sub $3, $3, $1
beq $0, $3, 1
j3
sw $4, 12 ($0)
```

Table 14 – Program to perform multiplication

Figure 29 shows the program as stored in the Instruction Memory of the simulation and Figure 30 the result of multiplying two and three.

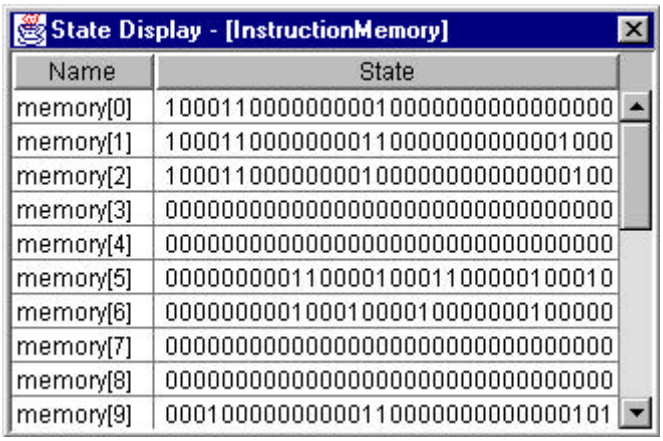

Figure 29 – Instruction memory containing multiplication program

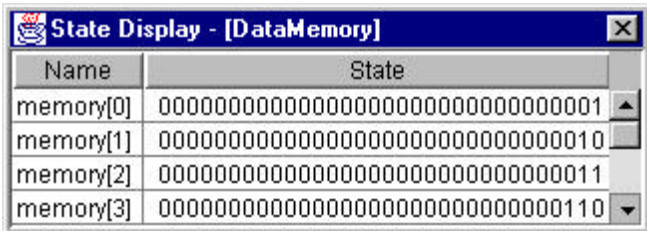

Figure 30 – Data memory containing result: 2 ´3=6

7.7 Critical Path

This design was chosen for its simplicity and is not very efficient. The CPU time taken to execute a given number of instructions is given by the following formula:

CPU Execution Time = Instruction Count × Cycles Per Instruction × Clock Cycle Time

Although we have only one clock cycle per instruction, the length of that clock cycle is large. A fixed clock cycle must be long enough to allow for the completion of the most complex instruction. The critical path for the load instruction includes an instruction fetch, a register access, an ALU operation, a memory access and another register access. As with most designs the length of each clock sub-cycle is equal so we have a total delay of $5 \times 186 = 930$.

One method of decreasing the execution time, that used in the design of CISC architectures, is to minimise the instruction count. A complex set of instructions is used, each of which performs a very specialised function. This tends to lead to a plethora of instruction formats and hence increased complexity in decoding the instruction to produce the required control lines. Care has to be taken to ensure that one of these complex instructions does not take longer to execute than a number of simpler instructions performing the same task.

8 Multi-Cycle Processor

As we are considering the MIPS RISC instruction set our aim is to reduce the product (Cycles Per Instruction × Clock Cycle Time). We shall set about this by breaking down the instruction execution into several stages as given in Table 15.

1	Instruction fetch
2	Instruction decode and register fetch
3	Execution, memory address calculation or branch computation
4	Memory access or R-type completion
5	Write back to register

Table 15 – Multi-cycle stages

Initially one might think that we have not gained anything. If we now have five cycles per instruction and a clock cycle time a fifth of that we had originally, then there is no difference. This is not the case however, only the load word instruction requires all five stages. A jump instruction, for example, only requires the first three stages. Therefore, the number of cycles per instruction is, on average, less than five and we have a decrease in execution time.

8.1 Datapath

The first things to notice looking at the datapath of Figure 31 is that there is now only one memory for both instructions and data and that we only have the one ALU to perform all the arithmetic operations. Any savings in silicon area such as these allow either a reduction in cost of the chip or space for other devices to increase speed, such as on-board cache.

1. In the first stage the PC address is used to fetch the next instruction from memory. The instruction is stored in a new register as the memory may be used again later in the instruction execution. The multiplexors at the ALU input are set up to add four to the PC and the result is written back into the PC.

2. The opcode passes onto the control unit for decoding while the data from the two registers is read. We also calculate the branch address by sign extending and shifting the lower sixteen bits of the instruction and then adding this to the PC. The ALU will be used for something else in the next stage so the address is stored in the target register. The data and/or branch address may not be used later but we do not lose anything by calculating them now and if they are needed, we gain.

3. If we are performing an R-format instruction the ALU multiplexors select the two data items and the ALU performs the required function. For a branch instruction the data is subtracted to decide whether the calculated target address is written to the PC. For a load or store instruction we add the sign extended offset to the address from the first register. Jump instructions are now completed by writing to the PC.

4. Load and store instructions write to, or read from, memory and R-format instructions write back to the destination register.

5. Load instructions write back to the required register.

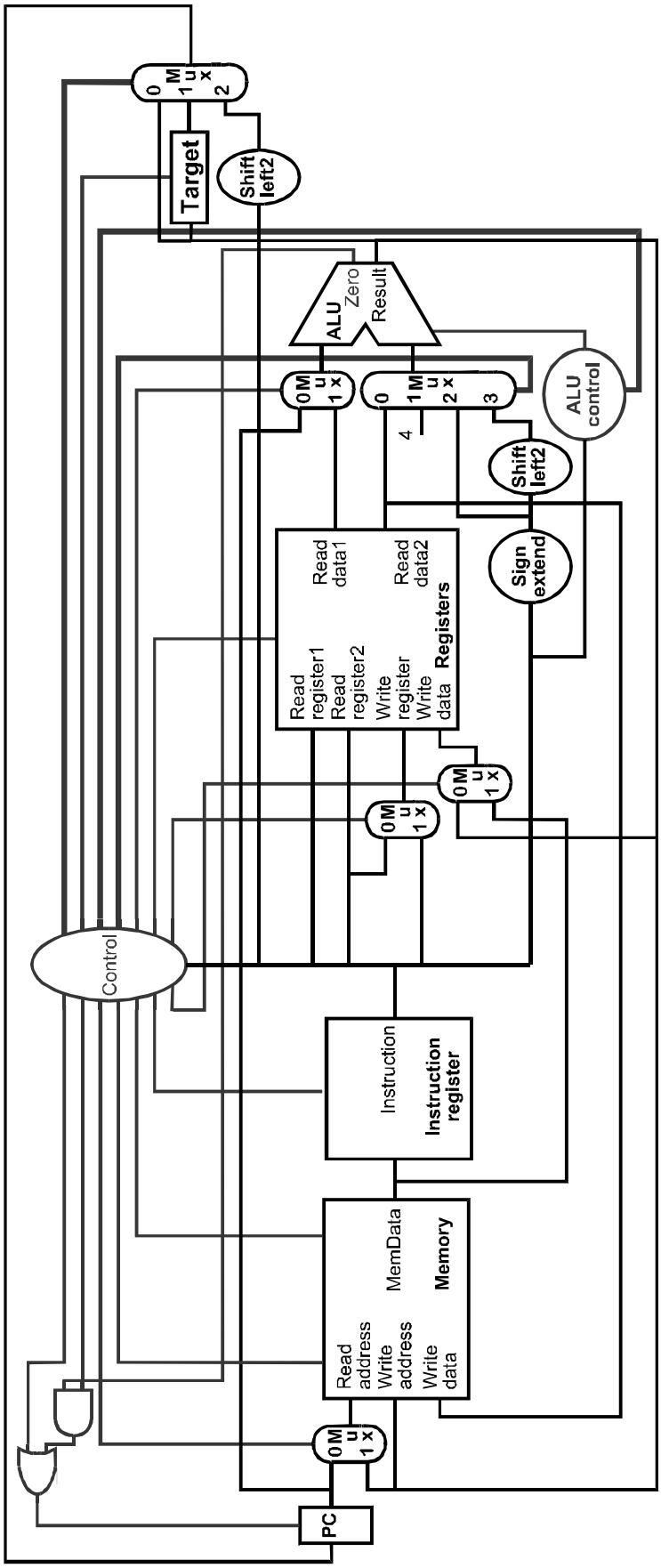

Figure 31 – Multi-cycle processor

8.2 Control Logic

As you might guess, the logic that is required to control the multi-cycle datapath is much more complicated than that for the single-cycle version. It is most easily represented by a flow chart, as illustrated in Figure 32. There are a number of ways of implementing this logic but all are based around the finite state machine shown in Figure 33. The inputs to the control logic are the opcode and current state and the outputs are the control lines and the next state. With ten states we require four bits to represent the state. There are seventeen control lines bringing the total number of outputs to twenty-one.

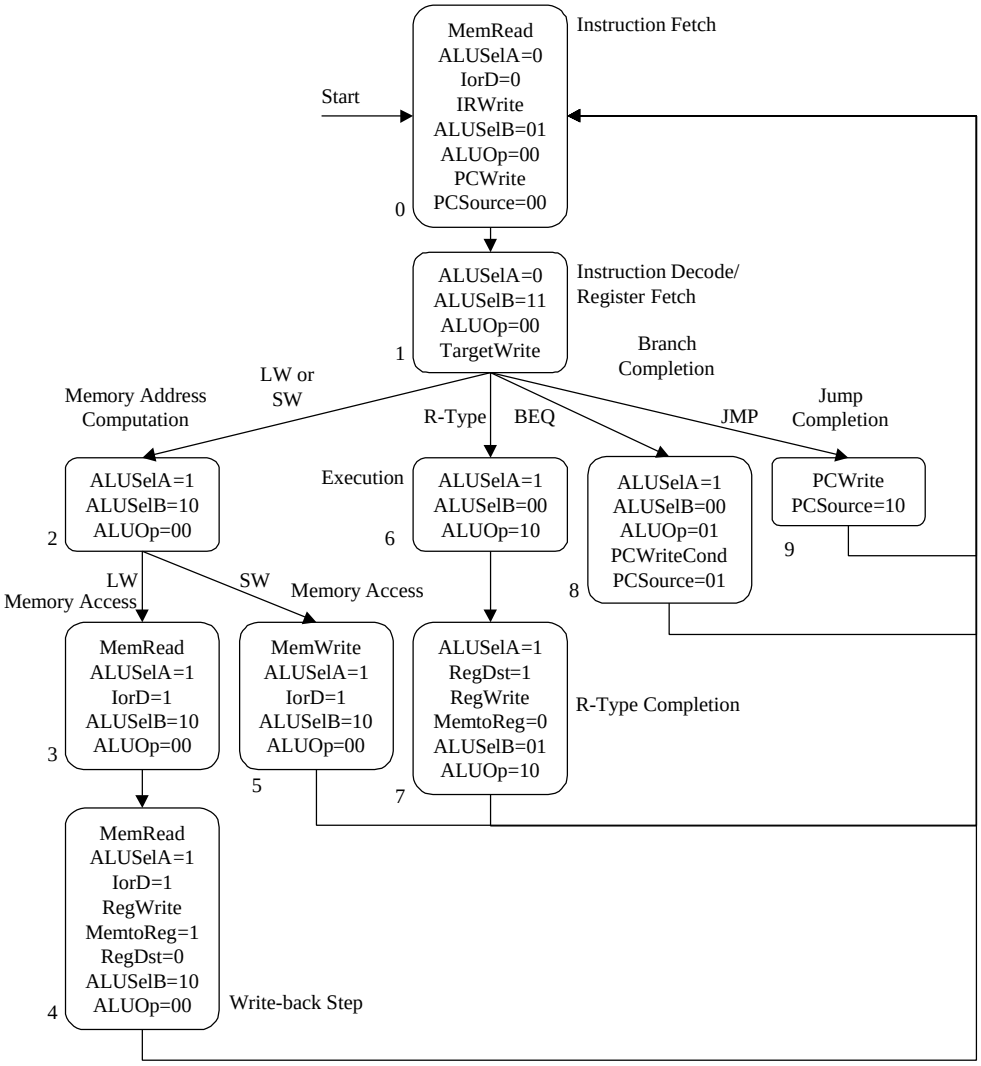

Figure 32 - Flow chart for control lines

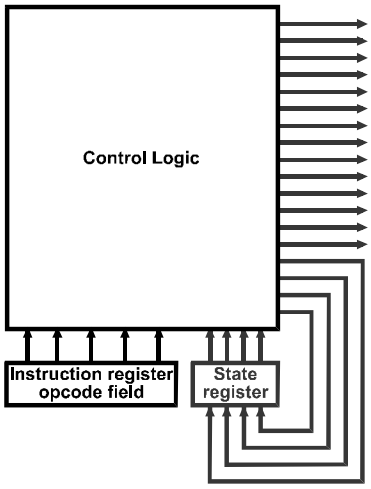

Figure 33 – Finite state machine

8.2.1 Read Only Memory (ROM)

The size of ROM required for a simplistic implementation would be $2^{10} \times (17 + 4) =$ 21Kbits. As we are only concerned with five of the possible 2^6 opcodes the truth table would be particularly sparse and hence the ROM mostly wasted. However, noting that the 17 control lines depend only on the current state, we can implement this part of the logic in $2^4 \times 17 =$ 272bits. The next state is dependent on both the current state and the opcode and this logic therefore requires $2^{10} \times 4 =$ 4096bits. Hence, by dividing the table in to two parts we can reduce the required space to only 4.3 Kbits. Even with this reduction there is still much wasted space. This is because many combinations of inputs and states never occur and often, as in state 0, we do not care what the opcode is.

8.2.2 Programmable Logic Array (PLA)

A PLA provides a more efficient implementation of a sparse truth table. The total size of a PLA is proportional to (#inputs × #product terms) + (#outputs × #product terms). For this application there are 18 product terms (combinations of opcode and state) therefore the size of the PLA is proportional to (10×18) + (21×18) = 558.

As with the ROM implementation it is possible to split the PLA into two. The PLA that outputs the control lines will require one product term for each state and therefore has a size proportional to (4×10) + (17×10) = 210. The second PLA has the remaining eight product terms and has size proportional to (10×8) + (4×8) = 112. This gives a total size proportional to 322 PLA cells, a huge improvement over the ROM implementation.

8.2.3 Explicit Next State

Although a PLA has given a compact implementation for our small instruction set, when we add more instructions the number of states and product terms will increase rapidly. In each of the implementations it can be seen that the majority of the control logic is used to calculate the next state. With a more complex instruction set we would see more sequences of states such as 2,3,4 in Figure 32. To take advantage of this we introduce a register holding the current state. Instead of outputs containing the next state we use a two-bit control code as shown in Table 16. The address select logic of Figure 34 uses this control code to control the next state. The two dispatch ROMs correspond to the decision points in states one and two. These take the opcode as input, and output the required next state.

AddrCtl value	Action
00	Set state to 0
01	Dispatch with ROM 1
10	Dispatch with ROM 2
11	Use the incremented state

Table 16- Address select lines

The circuit we now have in Figure 34 looks very much like the instruction fetch and branching of our processor. This leads to the concept of microcode. The state register becomes the microprogram counter and the PLA or ROM contains the microcode.

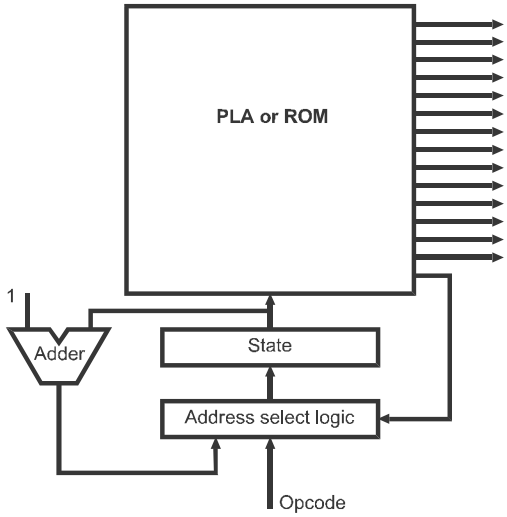

Figure 34 – Control logic with explicit next state

8.3 Simulation

The circuit of Figure 31 was entered into the simulator with an explicit next state control unit. The completed circuit can be seen in Figure 35. A two-phase clock is still required to ensure that the control lines are set up before data is transferred. The state of the control unit can be viewed as in Figure 36 to ensure that we progress through the control flow chart as required for the current instruction.

The same program as in the single-cycle case was run with the exception that the addresses of the data were changed to reflect their new location after the program code.

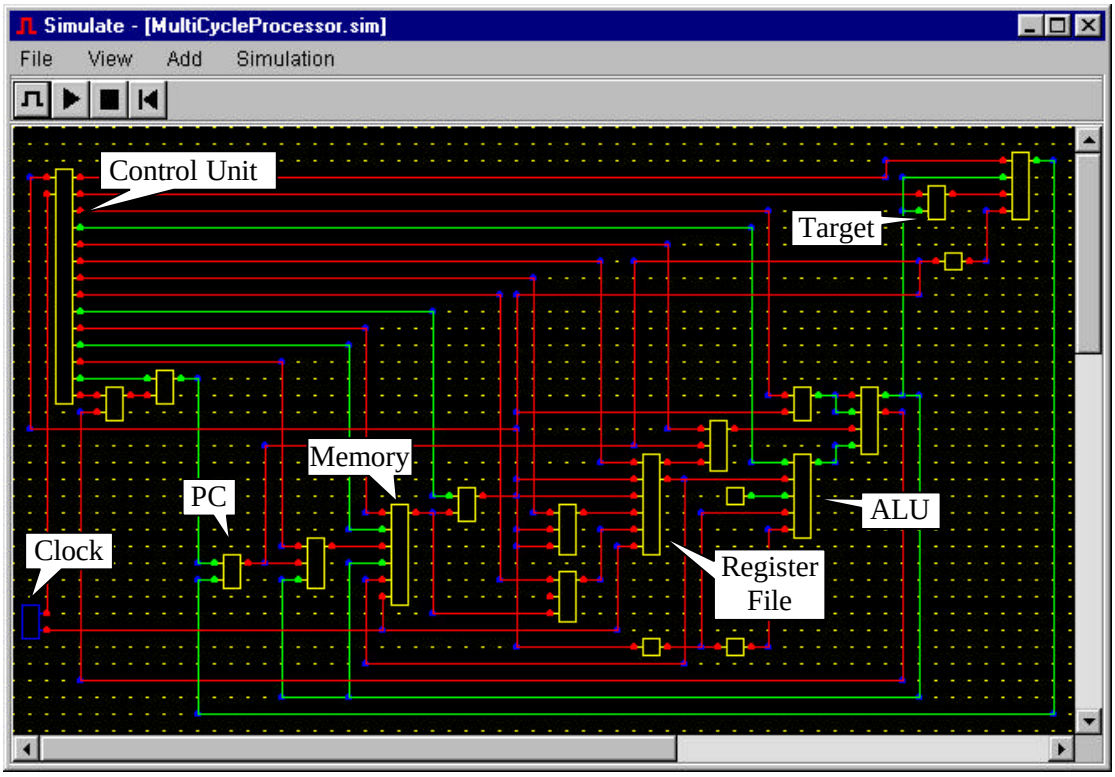

Figure 35 – Multi-cycle processor simulation

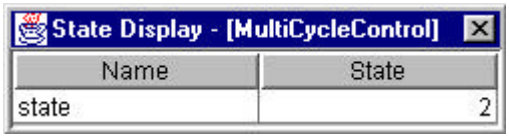

Figure 36 – State of control unit

The critical path for one clock cycle now has a delay of 192. Taking the distribution of instructions to be as in Table 17 results in an average of 4.07 clock cycles per instruction. The product (Cycles Per Instruction x Clock Cycle Time) is therefore now reduced to 781; a significant saving, particularly when considered in conjunction with the reduced hardware requirement.

Instruction	Frequency	Number of Clock Cycles
Load	32%	5
Store	17%	4
R-format	38%	4
Branch	10%	3
Jump	3%	3

Table 17 – Frequencies of instructions

9 Pipelined Processor

Having divided the instruction cycle into stages the processing of instructions takes place end to end as in Figure 37. A logical step would to fetch the second instruction when the first instruction moves onto the decode/register fetch stage. Similarly, when the first instruction is in the ALU stage and the second is being decoded, the third can be fetched. This process is know as pipelining and is illustrated in Figure 38. The advantage with this scheme is that, although each instruction still takes up to five cycles to complete, we begin a new instruction every clock cycle. Therefore, averaged over a large number of instructions, it is as if each instruction takes only one cycle.

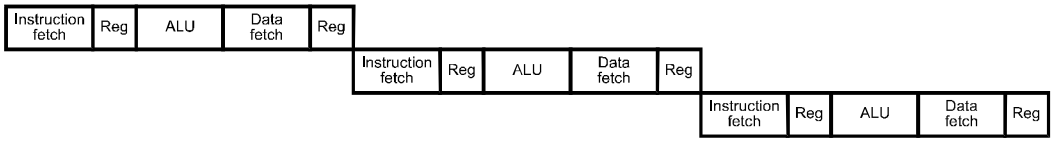

Figure 37 – Without pipelining

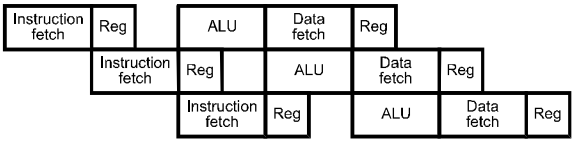

Figure 38 – With pipelining

As we hope to be carrying out each stage of the execution cycle simultaneously (although for different instructions) it makes sense to return to the single cycle architecture and build from there. The first thing to note is that the output from one stage will not stay constant while we calculate the next as it will have moved on to the next instruction. The answer is to insert a register between each of the five stages.

The control unit remains largely unchanged from the multi-cycle version. All that is required is to pass the control signals from stage to stage using the registers so that the controls signals for a particular instruction remain in step with the stage that is currently processing that instruction.

Care must be taken that the number of the required destination register reaches the registers at the same time as the data to be written. It must therefore be passed all the way through the intervening registers and back to the register file. The circuit can be seen in Figure 39.

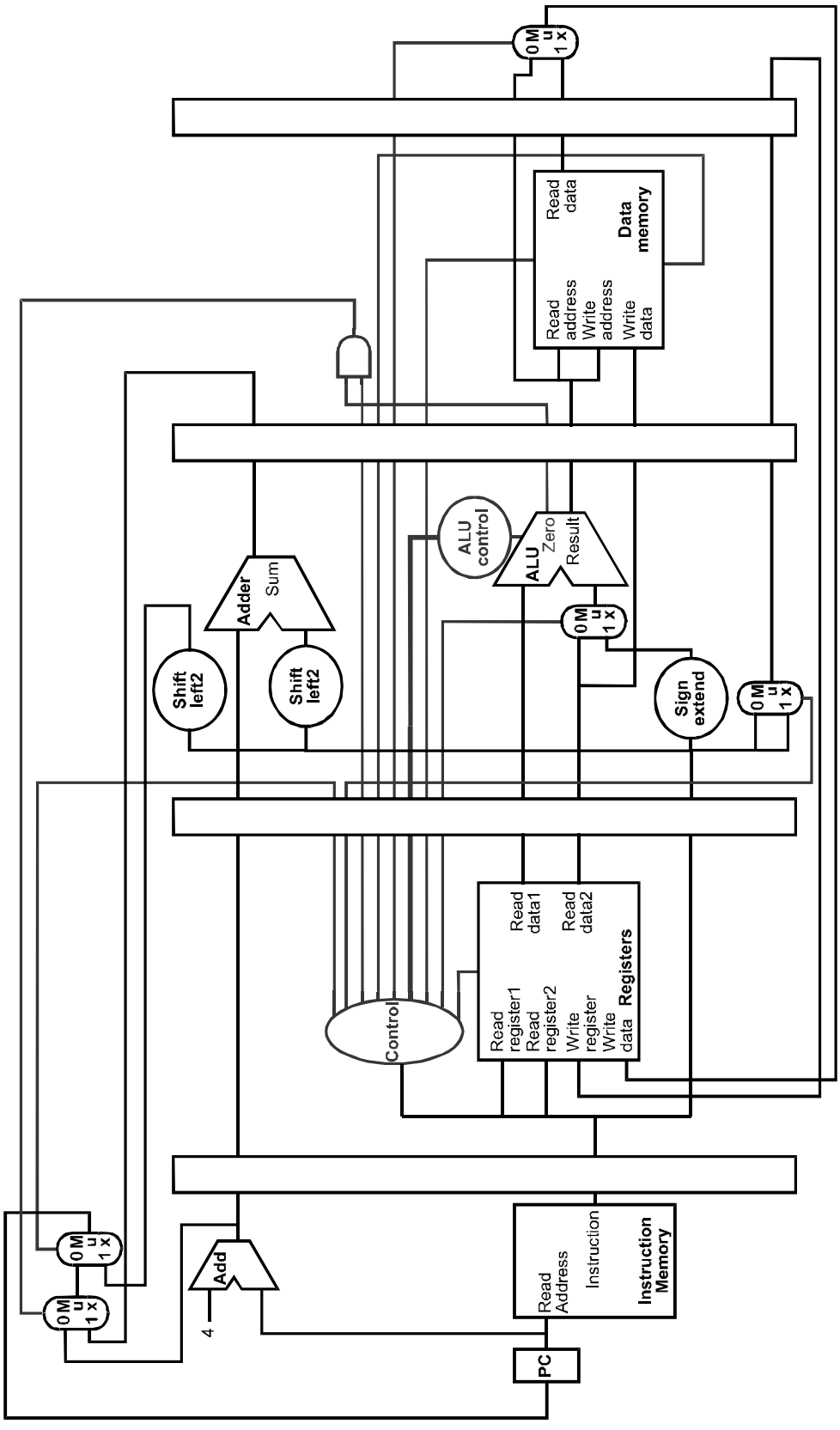

Figure 39 – Pipelined processor

9.1 Data Hazards

There are two problems with this circuit. The first is known as a data hazard. This occurs when an instruction writes to a register and a subsequent instruction attempts to read from that same register before the first instruction has finished writing to it. There are two possible solutions to this problem. The first is that the compiler should ensure that the situation does not arise. This could be done either by re-arranging the order of the statements and/or by adding no operation (nop) instructions between them. The other option is that the hardware detects data hazards and stalls the subsequent instructions until the hazard has cleared.

Inserting nop instructions and stalling the pipeline are obviously detrimental to the efficiency of the processor. This can be alleviated to some extent by a process known as internal forwarding. This relies on the fact that, although the answer to an operation has not yet been stored back into a register, it has still been calculated. Therefore, we can 'forward' the solution from its current position in the pipeline to where it is needed. Although clever, forwarding cannot help in the case of load instructions. In this case we must stall until the required memory location has been read.

9.2 Branch Hazards

The other type of hazard is due to branch instructions. What do we do with the instructions following a branch? If the branch does not take place we want to execute them, if not then we do not.

There are three options for dealing with this problem. Firstly, we could insert nop instructions as before. The second option is to always stall until we have decided whether to branch. This obviously results in the loss of several clock cycles. As Table 17 tells us that 10% of instructions are branches this is not going to be healthy for our total execution time. The third strategy is to begin by assuming that the branch will not be taken. If the branch does not take place we are on to a winner as we have already started to fetch and decode the next instructions. The disadvantage is the added complexity if the branch is required. In this case we must discard the instructions that follow the branch by setting all their control lines to zero.

9.3 Simulation

In order to keep the simulation simple the problems relating to hazards have been ignored. This places the responsibility on the 'compiler' to ensure that our programs do not cause data hazards by inserting nop instructions where required. The processor in mid-simulation can be seen in Figure 40.

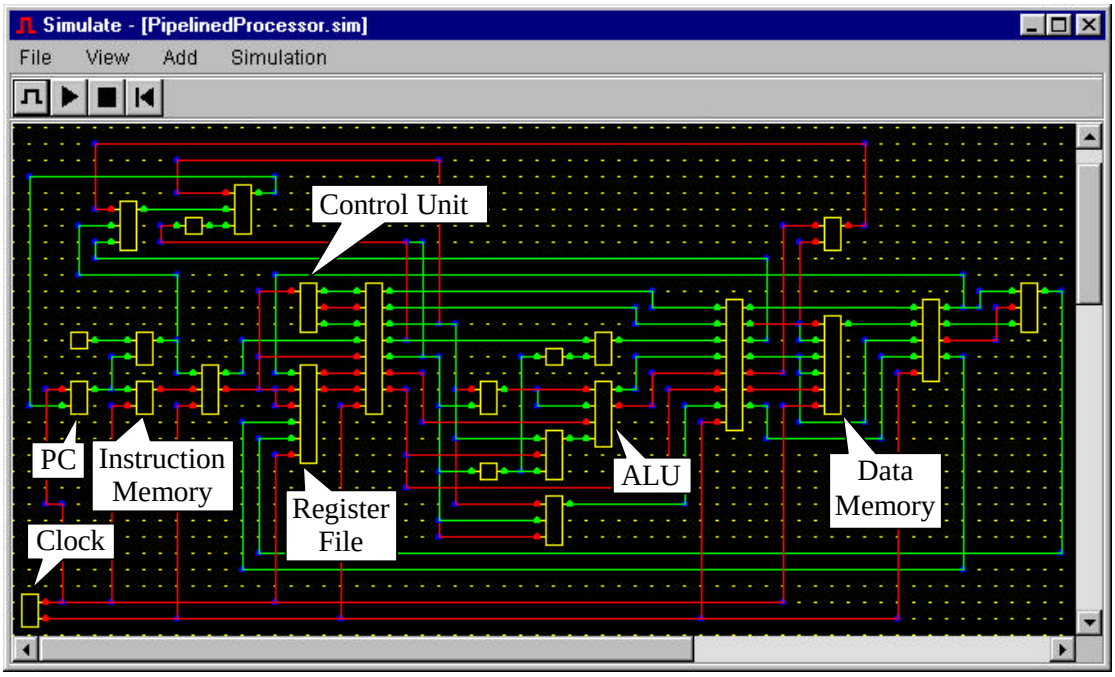

Figure 40 - Pipelined processor simulation

We must re-write our multiplication program to work on the new processor as shown in Table 18. Although we have more than doubled the length of the program you must bear in mind the five-fold increase in speed we achieve by using pipelining, the average time per instruction now being 184.

```
lw $1, 0 ($0)
lw $3, 8 ($0)
lw $2, 4 ($0)
nop
nop
sub $3, $3, $1
add $4, $2, $4
nop
nop
beq $0, $3, 1
nop
nop
nop
j 3
nop
nop
sw $4, 12 ($0)
```

Table 18 – Pipelined multiplication program

Note how, as well as inserting nop statements, we have re-ordered other statements. Loading registers one and three first allows us to do the subtraction one cycle earlier than we might otherwise have done. The addition instruction can in fact go in any of the three cycles between the subtraction and branch. If we had a program with a greater number of R-format instructions, as indicated by Table 17, then these could be used to fill in many of the gaps. Indeed, another method of reducing branch hazards is to place instructions that appear before the branch and do not affect the branch condition, in the spaces after the branch. It therefore does not matter whether or not the branch takes place as we require these instructions to be executed either way.

10 Conclusion

The main objective of the project, to design, build and test a general purpose, easy to use digital simulator was clearly met. This made the construction of circuits simple and their simulation straightforward to follow.

One problem remains: the speed of simulation. Replacing circuits with class files performing the same task was a simple way round this but it would have nice to have been able to look into packages during simulation to see exactly what was going on. Although it is possible that improvements could have been made to the simulator to increase its speed, the main difficulty is simply that Java is an interpreted language. We are therefore simulating a processor on a computer simulating a Java Virtual Machine. The use of a Just In Time (JIT) compiler for Windows 95 improved matters considerably. However, JIT compilers are at their most efficient when there are few objects which is not the case in these simulations. Maybe Sun's proposed 'HotSpot' compiler will deliver the desired performance.

There are a number of features that I would like to see added to the simulator. Firstly, it would be useful if the state of a package or circuit could be saved. For example, this would allow different programs to be loaded into the Instruction Memory.

The second enhancement would be to allow greater interaction during the simulation. One possibility would be the ability to change the values of unconnected inputs to permit the testing of components. If the state displays were changed to allow editing of the values (when the simulation is stopped) this would allow programs to be changed and registers such as the program counter to be set to particular values.

Another nice feature would be if the computer would calculate automatically the delay between a given input and output. The editing features could also be improved although this will be easier when the Swing Drag and Drop classes have been finalised.

The three architectures simulated during this project showed well the trade off between cycles per instruction and cycle length. It also illustrated the difficulties with schemes such as pipelining. There are innumerable ways in which the simulations could be extended to look at other areas of processor design. It would be a relatively simple matter to add the hazard detection features to the pipelined processor. Another major feature of RISC architectures, which we did not have time to look at here, is procedure call mechanisms such as the SPARC overlapping window registers.

In conclusion, the revised objectives of designing and building a digital simulation package and using it to investigate processor designs were fully met. In the process I have learnt many things. The background reading to the project cured some of my misconceptions regarding RISC processors. In future I shall think twice before using a programming language still in its infancy, and shy away from software beta releases. Another important lesson is to not be afraid to change your objectives as long as you have a clear reason for doing so. Having said this, looking back at the project, I cannot see much that I do would do differently.

11 References

Patterson & Hennessy, Computer Organization & Design : The Hardware / Software Interface, Morgan Kaufmann, 1994

Tanenbaum, Structured Computer Organization (Third Edition), Prentice Hall, 1990

Hennessy & Patterson, Computer Architecture : A Quantitative Approach (Second Edition), Morgan Kaufmann, 1996

Flanagan, Java in a Nutshell (Second Edition), O'Reilly, 1997

Hill & Peterson, Digital Logic and Microprocessors, Wiley, 1984

Clements, The Principles of Computer Hardware (Second Edition with corrections), Oxford University Press, 1992

Appendix A - Progress Report

A.1 Overview of Project

The aim of this project is to design, build and test a digital circuit simulator to allow the simulation of processor circuits. Specifically, it will be used to compare and contrast the major architectural differences between conventional Complex Instruction Set Computers (CISC) and Reduced Instruction Set Computers (RISC). The following RISC features will be examined in detail.

- Absence of micro-programming
- Single clock-cycle instructions
- Pipelining
- Overlapping register windows

The circuits will be built up from the logic gate level. Components will be combined into packages to provide a modular approach. Hopefully this will also enable a large degree of re-use between the two designs. However, to increase the speed of the processor simulation these modules may in some cases need to be written at a higher level of abstraction.

The decision was made to use the programming language Java to provide a simple means implementing the graphical interface and to allow portability over platforms.

A.2 Objectives

The objectives of the project are therefore defined as follows.

- To design and build a graphical digital circuit simulator
- To design a circuit for, and then simulate, a CISC processor
- To design a circuit for, and then simulate, a RISC processor
- To compare the architectures of these two processors using the simulations

A.3 Plan for Background Work

The background work for the project consists of two areas. Firstly, research into existing CISC and RISC architectures to allow the design of two representative circuits. Secondly, learning to use the latest version of Java (1.1.4) and assessing its suitability for this project. Both of these should be completed before the initial design phase takes place i.e. during the 1997 summer vacation.

A.4 Plan of Work and Milestones

The following table shows an outline plan of work with approximate deadlines. Many of the stages may proceed in parallel (e.g. the design of the two processor circuits).

Design digital simulator	0th Week Michaelmas
Code graphical user interface	2nd Week Michaelmas
Code loading, saving and placing of components	4th Week Michaelmas
Code circuit simulation	6th Week Michaelmas
Test simulation with simple circuits	8th Week Michaelmas
Design, enter and test circuit for CISC processor	0th Week Hilary
Design, enter and test circuit for RISC processor	4th Week Hilary
Compare and contrast CISC and RISC simulations	8th Week Hilary
Project write-up	0th Week Trinity

A.5 Interactions with Supervisor

As the objectives of the project are well defined and the route to reach them has been left up to me then meetings with my supervisor are only fortnightly during term-time. I have therefore met with Dr Sanders on the following dates – 27/6/97, 22/10/97, 7/11/97 and 21/11/97. These meetings have mainly been to monitor progress of the project and to discuss possible directions for future work.

.

A.6 Progress to Date

Progress to date has been much as outlined in the plan of work above. During the summer vacation research was carried out into the two architectures and a small prototype was built to assess the use of Java for the project. The design of the simulator was drawn up although possibly not in as much detail as I would have liked.

Work has continued to meet the deadlines given above. It is currently sixth week Michaelmas Term and the program is capable of simulating a simple two-bit ALU. As the project panned out the functionality of the simulator has been increased, for example in the ability to look into the contents of packages as they are simulated and in producing output traces from probes placed within the circuit. A great deal of time has also been put into producing a clear and easy-to-use graphical interface.

One minor problem has been the unveiling by Sun Microsystems of a Beta release for the Java Foundation Classes (Swing 0.5.1). Integrating this into the code required some re-writing and although useful in many areas the fact that it is only a Beta release is evident both in the number of bugs and lack of documentation. I am hoping that the first full release can be incorporated into the project before its completion.