9 Processor Design

Now, we are all set to design a basic processor for the *SimpleRisc* instruction set. We have assembled the arsenal to design a processor as well as making it efficient by our study of assembly languages, basic elements of a digital circuit (logical elements, registers, and memories), and computer arithmetic.

We shall start out by describing the basic elements of a processor in terms of fetching an instruction from memory, fetching its operands, executing the instruction, and writing the results back to memory or the register file. We shall see that along with the basic elements such as adders and multipliers, we need many more structures to efficiently route instructions between different units, and ensure fast execution. For modern processors, there are two primary design styles – hardwired, and microprogrammed. In the *hardwired* design style, we complicate a simple processor by adding more elaborate structures to process instructions. There is a net increase in complexity. In the microprogrammed design style, we have a very simple processor that controls a more complicated processor. The simple processor executes basic instructions known as microinstructions for each program instruction, and uses these microinstructions to control the operation of the complicated processor. Even though the hardwired design style is much more common today, the microprogrammed design style is still used in many embedded processors. Secondly, some aspects of microprogramming have crept into today's complex processors.

9.1 Design of a Basic Processor

9.1.1 Overview

The design of a processor is very similar to that of a car assembly line (see Figure 9.1). A car assembly line first casts raw metal into the chassis of a car. Then the engine is built, and put on the chassis. Then it is time to connect the wheels, the dashboard, and the body of the car. The final operation is to paint the car, and test it for manufacturing defects. The assembly

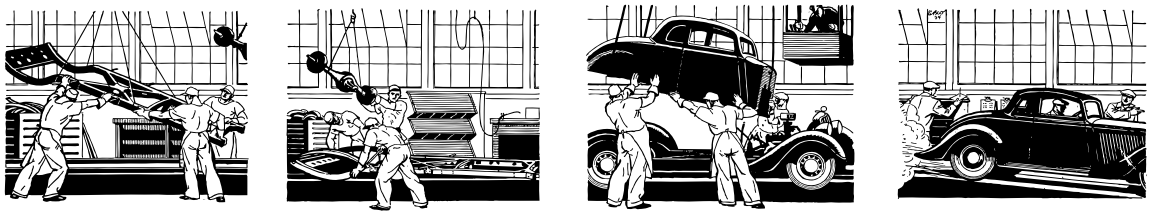

Figure 9.1: Car Assembly Line

line represents a long chain of actions, where one station performs a certain operation, and passes on a half built car to the next station. Each station also uses a pool of raw materials to augment the half built car such as metal, paint, or accessories.

We can think of a processor on the same lines as a car assembly line. In the place of a car, we have an instruction. The instruction goes through various stages of processing. The same way that raw metal is transformed to a beautiful finished product in a car factory, a processor acts upon a sequence of bits to do complex arithmetic and logical computations.

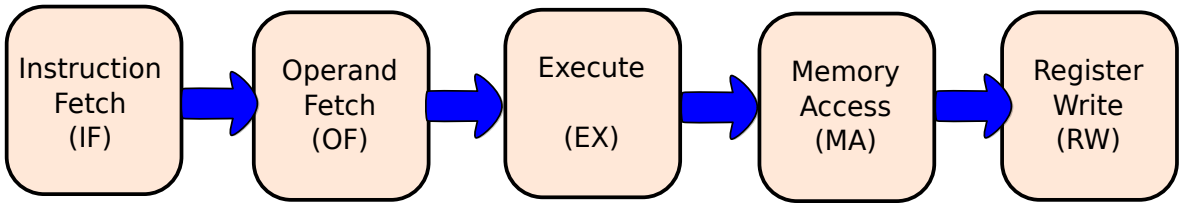

Figure 9.2: Five stages of instruction processing

We can broadly divide the operation of a processor into five stages as shown in Figure 9.2. The first step is to fetch the instruction from memory. The underlying organization of the machine does not matter. The machine can be a Von Neumann machine (shared instruction and data memory), or a Harvard machine (dedicated instruction memory). The fetch stage has logical elements to compute the address of the next instruction. If the current instruction, is not a branch, then we need to add the size of the current instruction (4 bytes) to the address stored in the PC. However, if the current instruction is a branch, then the address of the next instruction depends on the outcome and target of the branch. This information is obtained from other units in the processor.

The next stage is to "decode" the instruction and fetch its operands from registers. *SimpleRisc* defines 21 instructions, and the processing required for different instruction types is very different. For example, load-store instructions use a dedicated memory unit, whereas arithmetic instructions do not. To decode an instruction, processors have dedicated logic circuits that generate signals based on fields in the instruction. These signals are then used by other modules to properly process the instruction. The *SimpleRisc* format is very simple. Hence, decoding the instruction is very easy. However, commercial processors such as Intel processors have very elaborate decode units. Decoding the x86 instruction set is very complicated. Irrespective of

the complexity of decoding, the process of decoding typically contains the following steps – extracting the values of the operands, calculating the embedded immediate values and extending them to 32 or 64 bits, and generating additional information regarding the processing of the instruction. The process of generating more information regarding an instruction involves generating processor specific signals. For example, we can generate signals of the form "enable memory unit" for load/store instructions. For a store instruction, we can generate a signal to disable register write functionality.

In our *SimpleRisc* processor, we need to extract the immediate, and branch offset values embedded in the instruction. Subsequently, we need to read the values of the source registers. There is a dedicated structure in the processor called the register file that contains all the 16 *SimpleRisc* registers. For a read operation, it takes the number of the register as input, and produces the contents of the register as its output. In this step, we read the register file, and buffer the values of register operands in latches.

The next stage executes arithmetic and logical operations. It contains an arithmetic and logical unit(ALU) that is capable of performing all arithmetic and logical operations. The ALU is also required to compute the effective address of load-store operations. Typically, this part of the processor computes the outcome of branches also.

Definition 57
The ALU (arithmetic logic unit) contains elements for performing arithmetic and logical computations on data values. The ALU typically contains an adder, multiplier, divider, and has units to compute logical bitwise operations.

The next stage contains the memory unit for processing load-store instructions. This unit interfaces with the memory system, and co-ordinates the process of loading and storing values from memory. We shall see in Chapter 11 that the memory system in a typical processor is fairly complex. Some of this complexity is implemented in this part of the processor. The last step in processing an instruction is to write the values computed by the ALU or loaded values obtained from the memory unit to the register file.

9.2 Units in a Processor

9.2.1 Instruction Fetch – Fetch Unit

We start out by fetching an instruction from main memory. Recall that a *SimpleRisc* instruction is encoded as a sequence of 32 bits or 4 bytes. Hence, to fetch an instruction we need the starting address of the instruction. Let us store the starting address of the instruction in a register called the *program counter* (*pc*).

Important Point 13
Let us make an important distinction here between the terms PC and pc. PC is an acronym

for "program counter". In comparison, pc is a register in our pipeline, and will only be used to refer to the register, and its contents. However, PC is a general concept and will be used in the place of the term, "program counter", for the sake of brevity.

Secondly, we need a mechanism to update the PC to point to the next instruction. If the instruction is not a branch then the PC needs to point to the next instruction, whose starting address is equal to the value of the old PC plus 4 (REASON: each instruction is 4 bytes long). If the instruction is a branch, and it is taken, then the new value of the PC needs to be equal to the address of the branch target. Otherwise, the address of the next instruction is equal to the default value (current PC + 4).

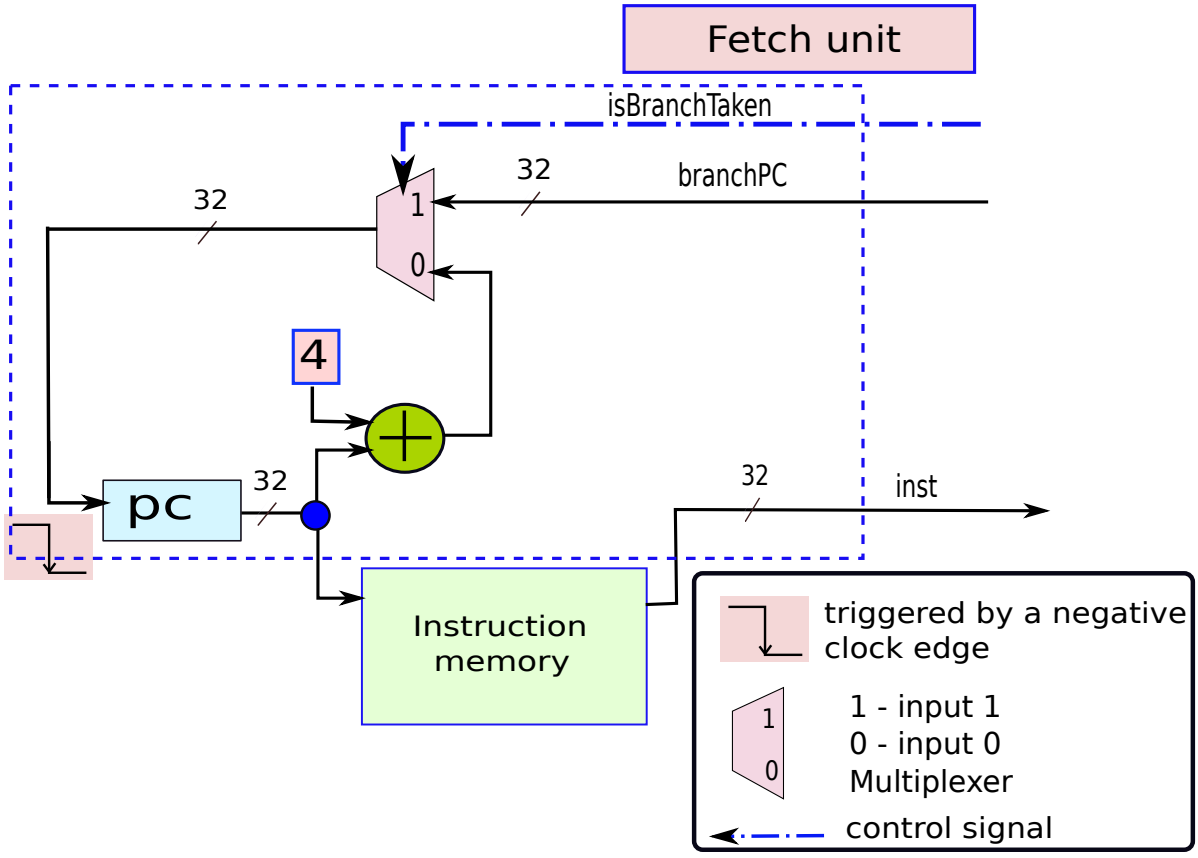

Figure 9.3: The Fetch Unit

Figure 9.3 shows an implementation of the circuit for the fetch unit. There are two basic operations that need to be performed in a cycle – (1) computation of the next PC, and (2) fetching the instruction.

The PC of the next instruction can come from two sources in *SimpleRisc* as shown in Figure 9.3. We can either use an adder and increment the current PC by 4, or we can get

the address from another unit that calculates the branch target($branchPC$), and the fact that the branch is taken. We can use a multiplexer to choose between these two inputs. Once, the correct input is chosen, it needs to be saved in the *pc* register and sent to the memory system for fetching the instruction. We can either use a combined memory for both instruction and data (Von Neumann Machine) or use a separate instruction memory (Harvard Machine). The latter option is more common. The instruction memory is typically implemented as an array of SRAM cells. The fetch unit provides the address in the SRAM array, and then uses the 32 bits stored at the specified starting address as the contents of the instruction.

Before proceeding to decode the instruction, let us make an important observation. Let us list down the external inputs of the fetch unit. They consist of the (1) branch target($branchPC$), (2) instruction contents, (3) and the signal to control the multiplexer ($isBranchTaken$). The branch target is typically provided by the decode unit, or the instruction execution unit. The instruction contents are obtained from the instruction memory. Let us now consider the case of the signal to control the multiplexer – $isBranchTaken$. The conditions for setting $isBranchTaken$ are shown in Table 9.1.

Instruction	Value of $isBranchTaken$
non-branch instruction	0
call	1
ret	1
b	1
beq	branch taken – 1 branch not taken – 0
bgt	branch taken – 1 branch not taken – 0

Table 9.1: Conditions for setting the $isBranchTaken$ signal

In our processor, a dedicated branch unit generates the $isBranchTaken$ signal. It first analyzes the instruction. If the instruction is a non-branch instruction, or a *call*/*ret*/*b* instruction, then the value of $isBranchTaken$ can be decided according to Table 9.1. However, if the instruction is a conditional branch instruction (*beq*/*bgt*), then it is necessary to analyze the flags register. Recall that the flags register contains the results of the last compare instruction (also see Section 3.3.2). We shall describe a detailed circuit for the branch unit in Section 9.2.4.

Let us refer to this stage of instruction processing as the IF (instruction fetch) stage. Before, we proceed to other stages, let us slightly digress here, and discuss two important concepts – *data path*, and *control path*.

9.2.2 Data Path and Control Path

We need to make a fundamental observation here. There are two kinds of elements in a circuit. The first type of elements are registers, memories, arithmetic, and logic circuits to process data values. The second type of elements are control units that decide the direction of the flow of data. The *control unit* in a processor typically generates signals to control all the multiplexers. These are called *control signals* primarily because their role is to control the flow of information.

We can thus conceptually think of a processor as consisting of two distinct subsystems. The first is known as the *data path* that contains all the elements to store and process information. For example the data memory, instruction memory, register file, and the ALU (arithmetic logic unit), are a part of the data path. The memories and register file store information, whereas the ALU processes information. For example, it adds two numbers, and produces the sum as the result, or it can compute a logical function of two numbers.

In comparison, we have a control path that directs the proper flow of information by generating signals. We saw one example in Section 9.2.1, where the control path generated a signal to direct a multiplexer to choose between the branch target and the default next PC. The multiplexer in this case was controlled by a signal $isBranchTaken$.

We can think of the control path and data path as two distinct elements of a circuit much like the traffic network of a city. The roads and the traffic lights are similar to the data path, where instead of instructions, cars flow. The circuits to control traffic lights constitute the control path. The control path decides the time of the transitions of lights. In modern smart cities, the process of controlling all the traffic lights in a city is typically integrated. A central command center controls all the lights and makes it possible to intelligently divert traffic to avoid traffic jams and accident sites. Similarly, a processor's control unit is fairly intelligent. Its job is to execute instructions as quickly as possible. In this book, we shall study a basic version of a control unit. However, the control unit will get very complicated in an advanced course on computer architecture.

Definition 58

Data Path *The data path consists of all the elements in a processor that are dedicated to storing, retrieving, and processing data such as register files, memories, and ALUs.*

Control Path *The control path primarily contains the control unit, whose role is to generate appropriate signals to control the movement of instructions, and data in the data path.*

A conceptual diagram showing the relationship between the control path and the data path is shown in Figure 9.4. After this short digression, let us now move on to discuss the next stage of instruction processing.

9.2.3 Operand Fetch Unit

SimpleRisc Instruction Format

Let us quickly recapitulate our knowledge about the *SimpleRisc* instruction format. The list of *SimpleRisc* instructions is shown in Table 9.2 along with their opcodes, and instruction format.

SimpleRisc is a simple and regular instruction set. It has three classes of instruction formats as shown in Table 9.3. The instruction formats are *branch*, *register*, and *immediate*. The *add*, *sub*, *mul*, *div*, *mod*, *and*, *or*, *cmp*, *not*, *lsl*, *lsr*, *asr*, and *mov* instructions can have either the

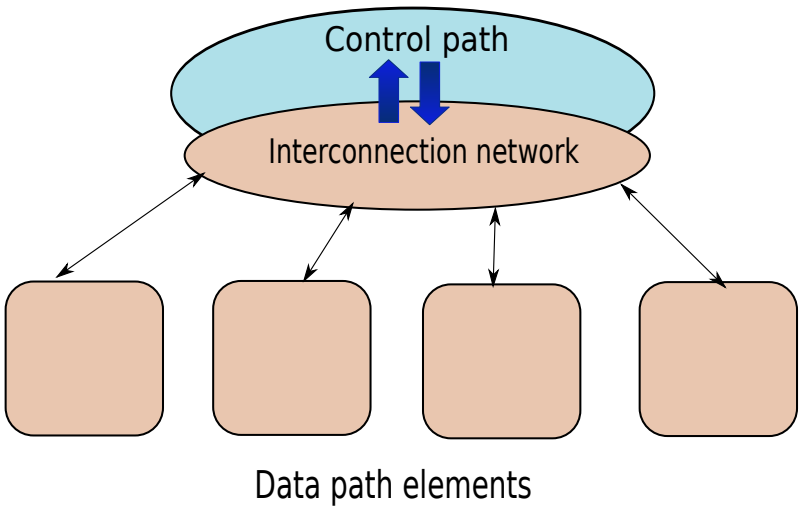

Figure 9.4: Relationship between the data path and control path

Inst.	Code	Format	Inst.	Code	Format
add	00000	add rd, rs1, (rs2/imm)	lsl	01010	lsl rd, rs1, (rs2/imm)
sub	00001	sub rd, rs1, (rs2/imm)	lsr	01011	lsr rd, rs1, (rs2/imm)
mul	00010	mul rd, rs1, (rs2/imm)	asr	01100	asr rd, rs1, (rs2/imm)
div	00011	div rd, rs1, (rs2/imm)	nop	01101	nop
mod	00100	mod rd, rs1, (rs2/imm)	ld	01110	ld rd, imm[rs1]
cmp	00101	cmp rs1, (rs2/imm)	st	01111	st rd, imm[rs1]
and	00110	and rd, rs1, (rs2/imm)	beq	10000	beq offset
or	00111	or rd, rs1, (rs2/imm)	bgt	10001	bgt offset
not	01000	not rd, (rs2/imm)	b	10010	b offset
mov	01001	mov rd, (rs2/imm)	call	10011	call offset
			ret	10100	ret

Table 9.2: List of instruction opcodes

Format	Definition				
branch	*op* (28-32)	$offset$ (1-27)			
register	*op* (28-32)	I (27)	rd (23-26)	$rs1$ (19-22)	$rs2$ (15-18)
immediate	*op* (28-32)	I (27)	rd (23-26)	$rs1$ (19-22)	imm (1-18)
$op \rightarrow$ opcode, $offset \rightarrow$ branch offset, $I \rightarrow$ immediate bit, $rd \rightarrow$ destination register					
$rs1 \rightarrow$ source register 1, $rs2 \rightarrow$ source register 2, $imm \rightarrow$ immediate operand					

Table 9.3: Summary of instruction formats

register or the *immediate* format. This is decided by the I bit (27^{th} bit) in the instruction. The *cmp* instruction does not have a destination register. The *mov* and *not* instructions have only one source operand. For further details, the reader can refer to Table 9.2, or Section 3.3.

The Operand Fetch Unit

The operand fetch unit has two important functions – (1) calculate the values of the immediate operand and the branch target by unpacking the offset embedded in the instruction, and (2) read the source registers.

Computation of the Immediate Operand and the Branch Target

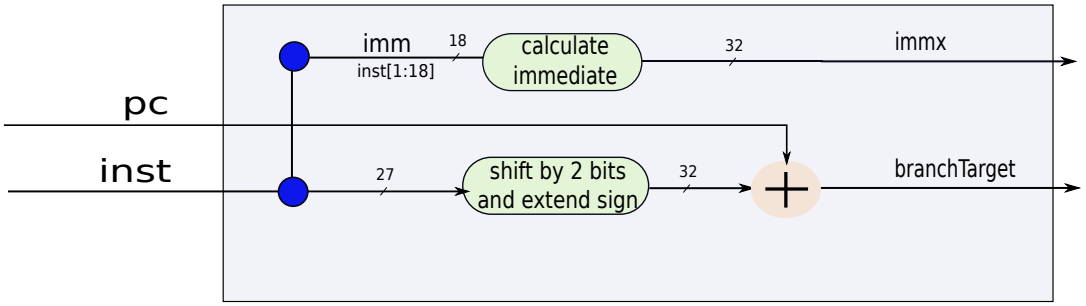

Figure 9.5: Calculation of the immediate operand and the branch target

Figure 9.5 shows the circuit for calculating the immediate operand, and the branch target. To calculate the immediate operand, we need to first extract the *imm* field (bits 1-18) from the instruction. Subsequently, we extract the lower 16 bits, and create a 32-bit constant in accordance with the modifiers (bit 17, and 18). When no modifier is specified, we extend the sign of the 16-bit number to make it a 32-bit number. For the u modifier, we fill the top 16 bits with 0s, and for the h modifier, we shift the 16-bit number, 16 positions to the left. The newly constructed 32-bit value is termed as $immx$.

Similarly, we can compute the signal, $branchTarget$ (branch target for all types of branches excluding *ret*). We need to first extract the 27 bit offset (bits 1 to 27) from the instruction. Note that these 27 bits represent the offset in terms of memory words as described in Section 3.3.14. Thus, we need to shift the offset to the left by 2 bits to make it a 29 bit number, and then extend its sign to make it a 32-bit number. Since we use PC-relative addressing in *SimpleRisc* , to obtain the branch target we need to add the shifted offset to the PC. The branch target can either be derived from the instruction ($branchTarget$ signal), as we have just described, or in the case of a *ret* instruction, the branch target is the contents of the *ra* register. In this case, the *ra* register comes from the register file. We choose between both the values in the next stage, and compute $branchPC$.

There is a need to make an important observation here. We are calculating $branchTarget$ and $immx$ for all instructions. However, any instruction in the *SimpleRisc* format will only require at the most one of these fields ($branchTarget$ or $immx$). The other field will have junk

values. Nevertheless, it does not hurt to pre-compute both the values in the interest of speed. It is necessary to ensure that the correct value is used in the later stages of processing.

Reading the Registers

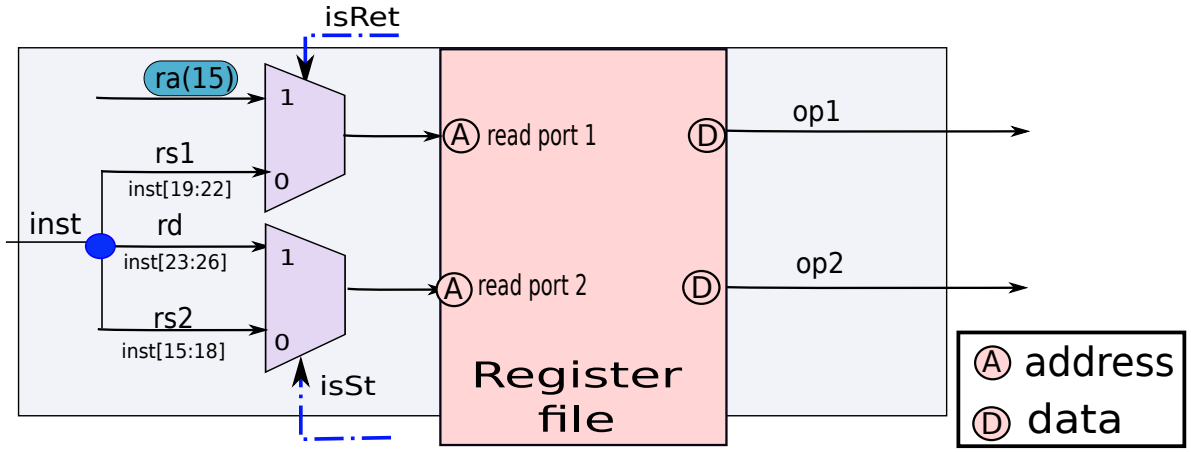

Figure 9.6: Reading the Source Registers

In parallel, we can read the values of the source registers as shown. Here, also we follow the same strategy. We read more than what we require. Critics might argue that this approach wastes power. However, there is a reason for doing so. Extra circuitry is required to decide if a given operand is actually required. This has an adverse impact in terms of area, and time. The operand fetch unit becomes slower. Hence, we prioritize the case of simplicity, and read all the operands that might be required.

The circuit for reading the values of source registers is shown in Figure 9.6. The register file has 16 registers, two read ports, and one write port (not shown in the figure). A *port* is a point of connection (an interface) in a hardware structure, and is used for the purpose of either entering inputs, or reading outputs. We can have a read port (exclusively for reading data), a write port (exclusively for writing data), and a read-write port (can be used for both reading and writing).

Definition 59
A port *is a point of connection in a hardware structure, and is used for the purpose of either entering inputs, or reading outputs. We can have a read port (exclusively for reading data), a write port (exclusively for writing data), and a read-write port (can be used for both reading and writing).*

For the first register operand, $op1$, we have two choices. For ALU, and memory instructions, we need to read the first source register, $rs1$ (bits 19 to 22). For the *ret* instruction, we need to

read the value of the return address register, ra. To choose between the contents of the field, $rs1$, in the instruction and ra, we use a multiplexer. The multiplexer is controlled by a signal, $isRet$. If $isRet$ (is return) is equal to 1, then we choose ra, otherwise we choose $rs1$. This value is an input to the register file's first read port. We term the output of the first read port as $op1$ (operand 1).

We need to add a similar multiplexer for the second read port of the register file too. For all the instructions other than the store instruction, the second source register is specified by the $rs2$ (bits 15 to 18) field in the instruction. However, the store instruction is an exception. It contains a source register in rd (bits 23 to 26). Recall that we had to make this bitter choice at the cost of introducing a new instruction format. Since we have a very consistent instruction format (see Table 9.3) the process of decoding is very simple. To extract different fields of the instruction ($rs1$, $rs2$, opcode, and imm) we do not need additional logic elements. We need to save each bit of the instruction in a latch, and then route the wires appropriately.

Coming back to our original problem of choosing the second register operand, we observe that we need to choose the right source register – $rs2$ or rd. The corresponding multiplexer is controlled by the $isSt$ (is store) signal. We can quickly find out if the instruction is a store by using a set of logic gates to verify if the opcode is equal to 01111. The result of the comparison is used to set the $isSt$ signal. The corresponding output of the register file is termed as $op2$ (operand 2).

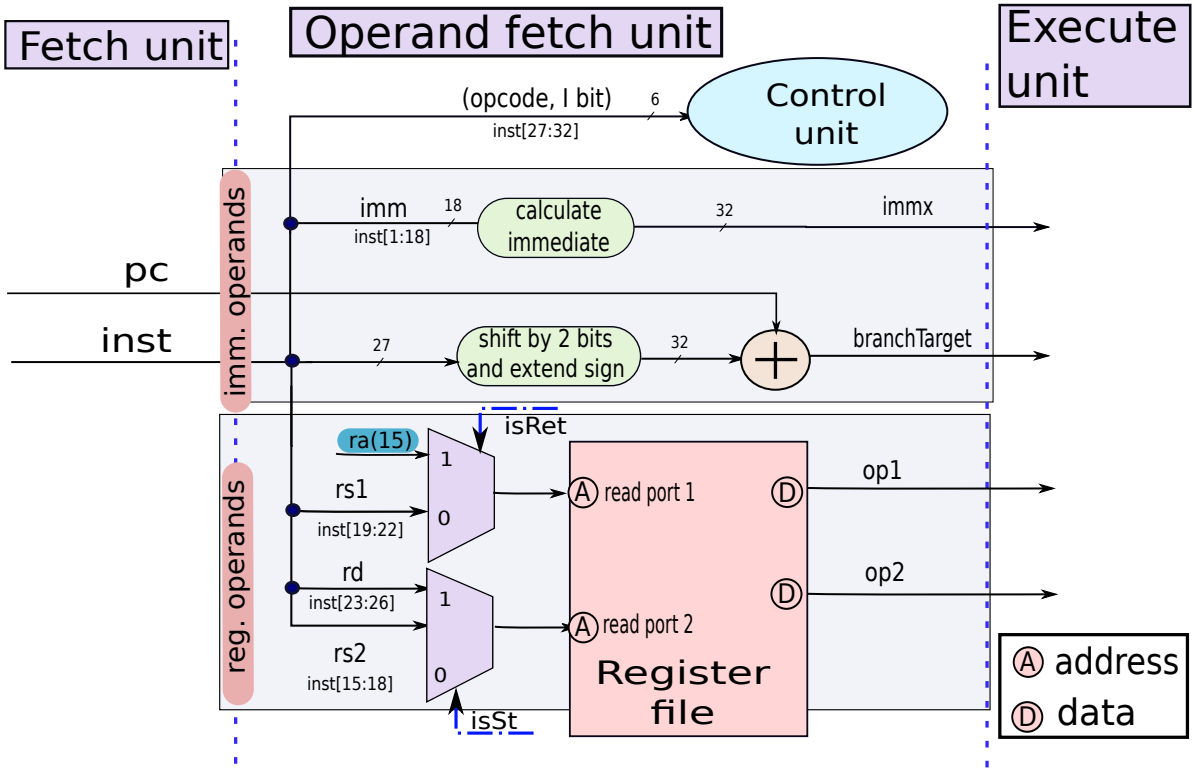

Figure 9.7: Operand Fetch Stage

Lastly, it is necessary to send the opcode (5 bits), and the immediate bit (1 bit) to the control unit such that it can generate all the control signals. The complete circuit for the operand fetch unit is shown in Figure 9.7. $op1$, $op2$, $branchTarget$, and $immx$ are passed to the execute unit.

9.2.4 Execute Unit

Let us now look at executing an instruction. Let us start out by dividing instructions into two types – branch and non-branch. Branch instructions are handled by a dedicated branch unit that computes the outcome, and final target of the branch. Non branch instructions are handled by an ALU (arithmetic logic unit).

Branch Unit

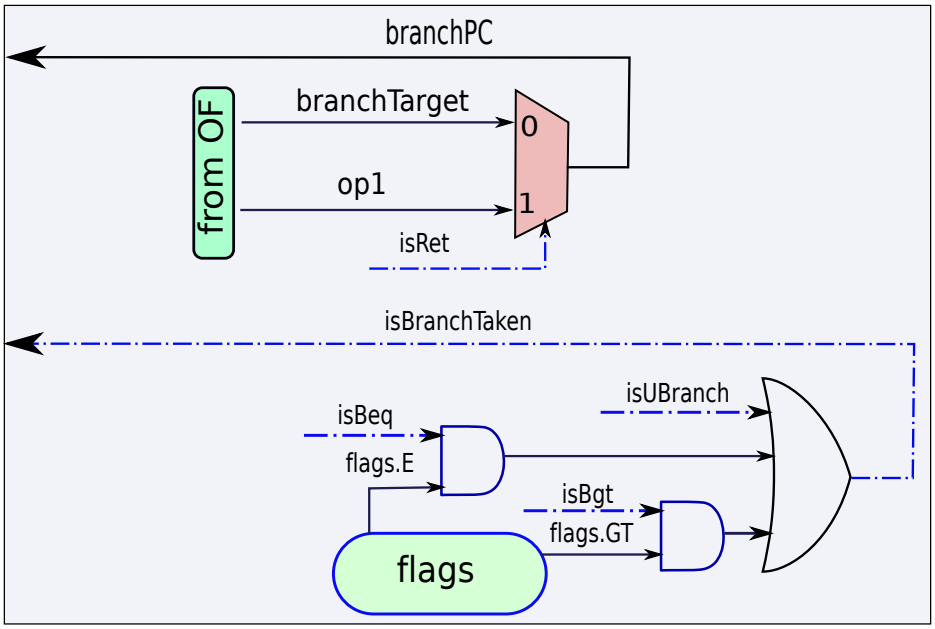

Figure 9.8: Branch Unit

The circuit for the branch unit is shown in Figure 9.8.

First, we use a multiplexer to choose between the value of the return address ($op1$), and the $branchTarget$ embedded in the instruction. The $isRet$ signal controls the multiplexer. If it is equal to 1, we choose $op1$; otherwise, we choose $branchTarget$. The output of the multiplexer, $branchPC$, is sent to the fetch unit.

Now, let us consider the circuit to compute the branch outcome. As an example, let us consider the case of the *beq* instruction. Recall that the *SimpleRisc* instruction set requires a $flags$ register that contains the result of the last compare (cmp) instruction. It has two bits – E and GT. If the last compare instruction led to an equality, then the E bit is set, and

if the first operand was greater than the second operand then the GT bit is set. For the *beq* instruction, the control unit sets the signal $isBeq$ to 1. We need to compute a logical AND of this signal and the value of the E bit in the $flags$ register. If both are 1, then the branch is taken. Similarly, we need an AND gate to compute the outcome of the *bgt* instruction, as shown in Figure 9.8. The branch might also be unconditional ($call/ret/b$). In this case, the control unit sets the signal $isUBranch$ to 1. If any of the above conditions is true, then the branch is taken. We subsequently use an OR gate that computes the outcome of the branch, and sets the $isBranchTaken$ signal. This signal is used by the fetch unit to control the multiplexer that generates the next PC.

ALU

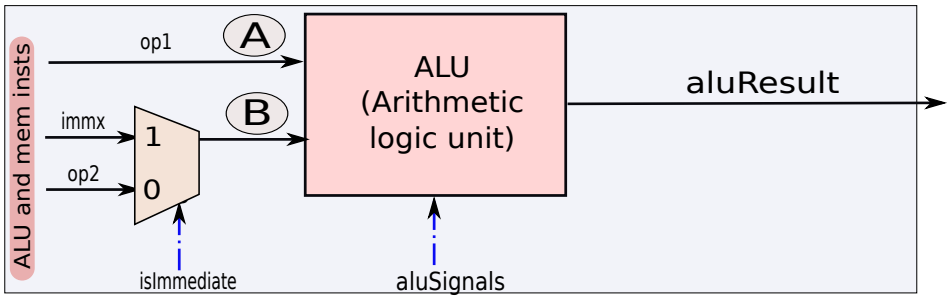

Figure 9.9: ALU

Figure 9.9 shows the part of the execution unit that contains the ALU. The first operand (A) of the ALU is always $op1$ (obtained from the operand fetch unit). However, the second operand (B) can either be a register or the sign extended immediate. This is decided by the $isImmediate$ signal generated by the control unit. The $isImmediate$ signal is equal to the value of the immediate bit in the instruction. If it is 1, then the multiplexer in Figure 9.9 chooses $immx$ as the operand. If it is 0, then $op2$ is chosen as the operand. The ALU takes as input a set of signals known collectively as $aluSignals$. They are generated by the control unit, and specify the type of ALU operation. The result of the ALU is termed as $aluResult$.

Figure 9.10 shows the design of the ALU. The ALU contains a set of modules. Each module computes a separate arithmetic or logical function such as addition or division. Secondly, each module has a dedicated signal that enables or disables it. For example, there is no reason to enable the divider when we want to perform simple addition. There are several ways in which we can enable or disable a unit. The simplest method is to use a transmission gate for every input bit. A transmission gate is shown in Figure 9.11. If the signal(S) is turned on, then the output reflects the value of the input. Otherwise, it maintains its previous value. Thus, if the enabling signal is off, then the module does not see the new inputs. It thus does not dissipate any power, and is effectively disabled.

Let us consider each of the modules in the ALU one after another. The most commonly used module is the adder. It is used by *add*, *sub*, and *cmp* instructions, as well as by load and store instructions to compute the memory address. It takes A and B as inputs. Here, A and B are the values of the source operands. If the $isAdd$ signal is turned on, then the adder adds the

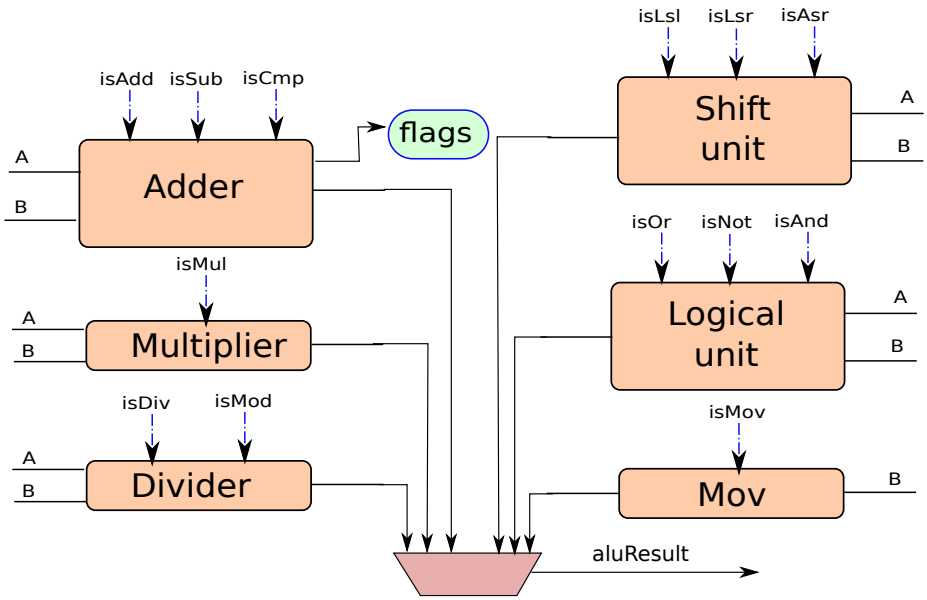

Figure 9.10: ALU

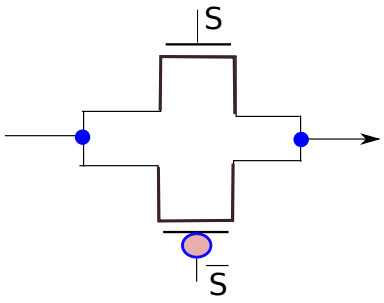

Figure 9.11: A transmission gate

operands. Likewise, if the $isSub$ signal is turned on, then the adder adds the 2's complement of B with A. In effect, it subtracts B from A. If the $isCmp$ flag is turned on, then the adder unit subtracts B from A and sets the value of the $flags$ register. If the output is 0, then it sets the E bit. If the output is positive, it sets the GT bit. If none of these signals ($isAdd/isSub/isCmp$) is true, then the adder is disabled.

The multiplier and divider functions are implemented similarly. The multiplier is enabled by the $isMul$ signal, and the divider is enabled by the $isDiv$ or $isMod$ signals. If the $isDiv$ signal is true, then the result is the quotient of the division, whereas, if the $isMod$ signal is true, the result is the remainder of the division.

The shift unit left shifts, or right shifts A, by B positions. It takes three signals – $isLsl$, $isLsr$, and $isAsr$. The logical unit consists of a set of AND, OR, and NOT gates. They are enabled by the signals $isOr$, $isAnd$, and $isNot$, respectively. The Mov unit is slightly special

in the sense that it is the simplest. If the $isMov$ signal is true, then the output is equal to B. Otherwise, it is disabled.

To summarize, we show the full design of the execution unit (branch unit and ALU) in Figure 9.12. To set the output ($aluResult$), we need a multiplexer that can choose the right output out of all the modules in the ALU. We do not show this multiplexer in Figure 9.12. We leave the detailed design of the ALU circuit along with the transmission gates and output multiplexer as an exercise for the reader.

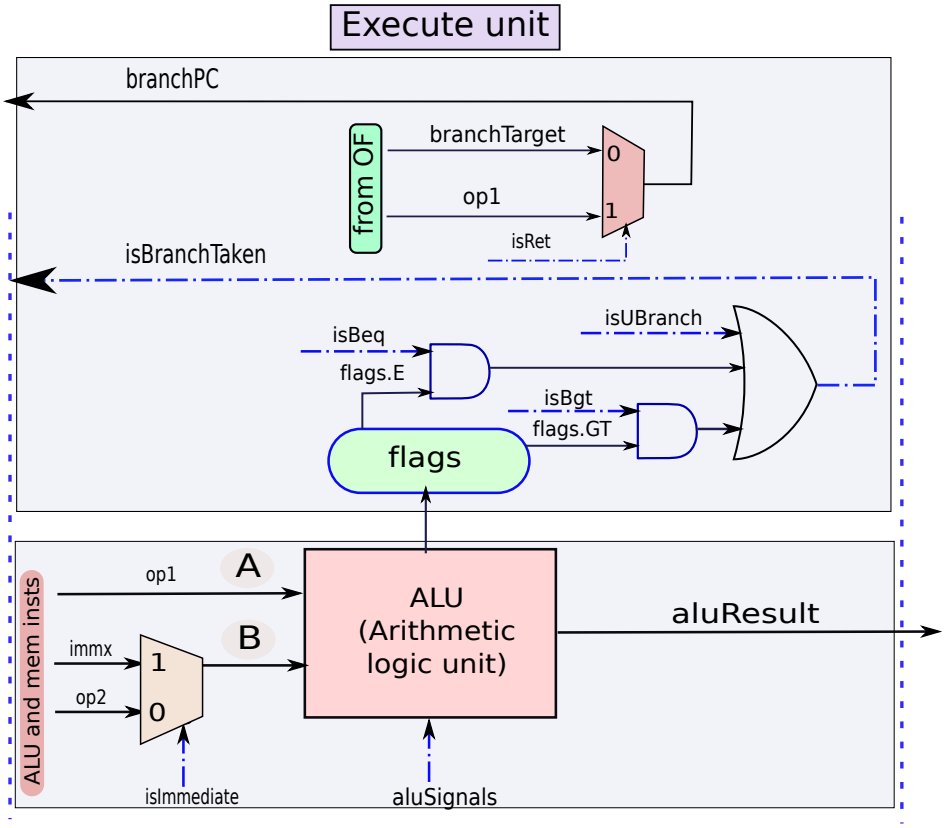

Figure 9.12: Execute Unit (Branch and ALU unit)

9.2.5 Memory Access Unit

Figure 9.13 shows the memory access unit. It has two inputs – data and address. The address is calculated by the ALU. It is equal to the result of the ALU ($aluResult$). Both the load and store instructions use this address. The address is saved in a register traditionally known as MAR (memory address register).

Let us now consider the case of a load instruction. In *SimpleRisc* , the format of the load instruction is *ld rd, imm[rs1]*. The memory address is equal to the immediate value plus the contents of register, $rs1$. This is the value of $aluResult$. The memory unit does not require any

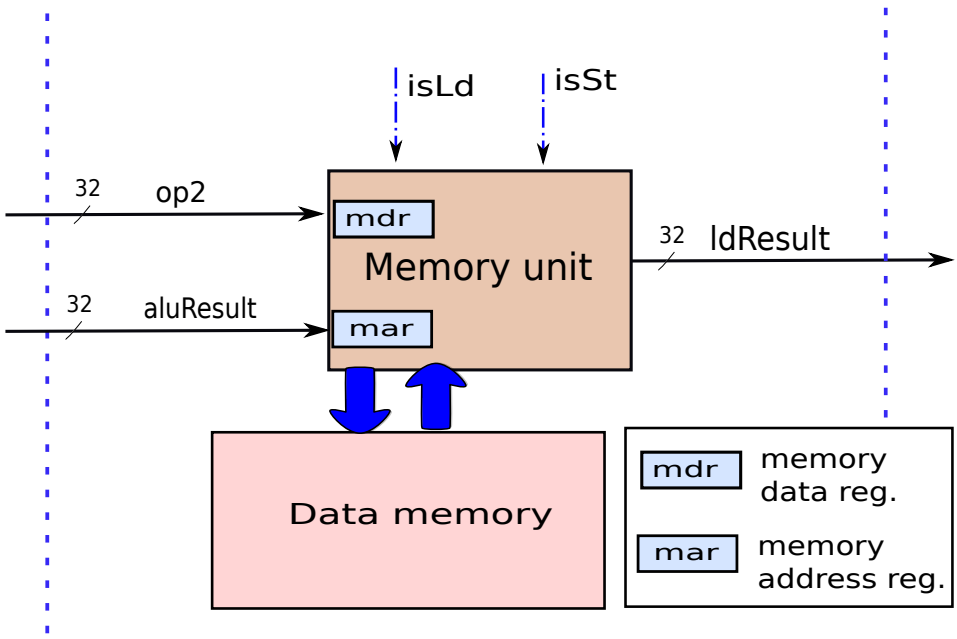

Figure 9.13: Memory Unit

other inputs. It can proceed to fetch the value of the memory address from the data memory. The memory unit reads 4 bytes starting from the memory address. The result (*ldResult*) is now ready to be written to the destination register.

The format of the store instruction is : *st rd, imm[rs1]*. The address is computed using the ALU similar to the way the address is calculated for the load instruction. For the store instruction, *rd* is a source register. The contents of register *rd* ($reg[rd]$) are read by the operand fetch unit (see Section 9.2.3). This value is termed as *op*2 in Figure 9.6. *op*2 contains the contents of register *rd*, and represents the data of the store instruction. The memory unit writes the value of *op*2 to the MDR (memory data register) register. In parallel, it proceeds to write the data to data memory. The store instruction does not have an output.

Note that here also we follow the same naming scheme as we had followed for PC and *pc*. MAR is an acronym for (memory address register), whereas *mar* refers specifically to the *mar* register in the data path.

Now, the memory unit takes two control signals as inputs – $isLd$, and $isSt$. For obvious reasons, at most one of these signals can be true at any one time. If none of these signals is true, then the instruction is not a memory instruction, and the memory unit is disabled.

A subtle point needs to be discussed here. MAR and MDR are traditionally referred to as registers. However, they are not conventional edge triggered registers. They are used like temporary buffers that buffer the address and the store values till the memory request completes.

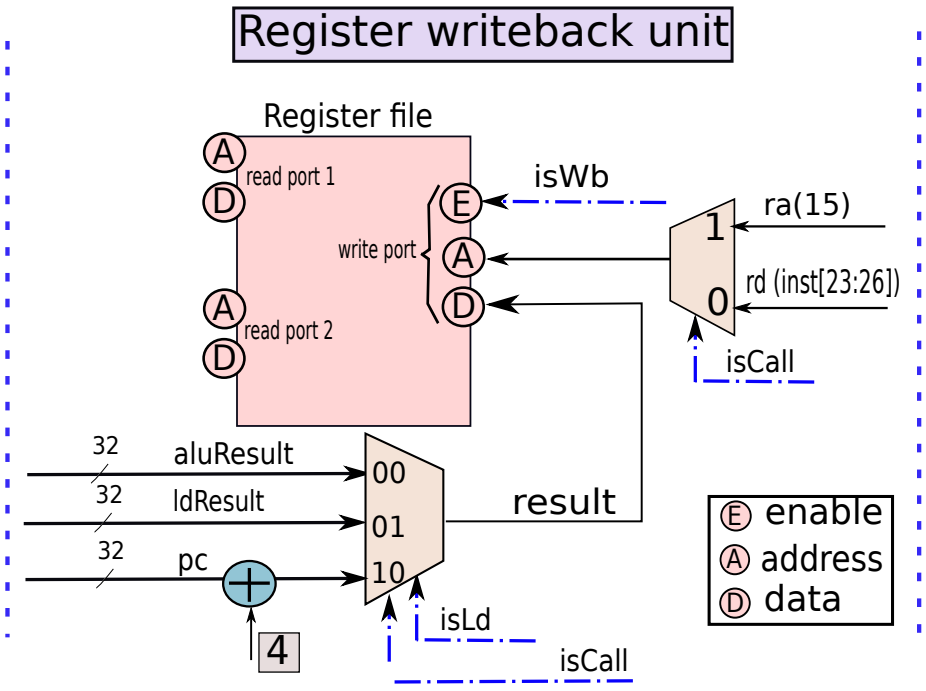

Figure 9.14: Register Writeback Unit

9.2.6 Register Writeback Unit

The last step of instruction processing is to write the computed values back to the register file. This value can be the output of a load or ALU instruction, or the return address written by the *call* instruction. This process is known as writeback, or register writeback. We refer to this unit as the *register writeback(RW) unit*. Its circuit diagram is shown in Figure 9.14.

We first need to choose the right source operand. We have three choices – *aluResult*, *ldResult*, or the return address. The return address is equal to the PC of the *call* instruction plus 4. We use a multiplexer to choose between the three input values. We use two control signals to control the multiplexer. The first control signal is $isLd$ (is load), and the second control signal is $isCall$. We choose *aluResult*, when both the control signals are 0. We choose *ldResult*, when $isLd = 1$, and $isCall = 0$, and lastly, we choose $PC + 4$, when $isCall$ is equal to 1. The output of the multiplexer, *result*, needs to be written to the register file.

Note that we had shown a partial view of the register file when we discussed the operand fetch unit in Section 9.2.3. We showed only two read ports. However, the register file also has a write port that is used to write data. The write port has three inputs – address, data, and enable bit. The address is either the number of the destination register *rd* or the id of the return address register (15). The correct address needs to be chosen with a multiplexer. The destination register is specified by bits 23 to 26 of the instruction. The second multiplexer chooses the data that needs to be written. The output of this multiplexer is sent to the data pins of the write port. Lastly, we need an enable signal ($isWb$) that specifies if we need to write the value of a register. For example, the store, *nop*, and compare instructions do not need a

register writeback. Hence, for these instructions, the value of $isWb$ is false. It is also false for branch (excluding *call*) instructions. $isWb$ is true for the rest of the ALU instructions, *mov* and *ld* instructions.

9.2.7 The Data Path

Let us now form the whole by joining all the parts. We have up till now divided a processor into five basic units: instruction fetch unit (IF), operand fetch unit (OF), execution unit (EX), memory access unit (MA), and register writeback unit (RW). It is time to combine all the parts and look at the unified picture. Figure 9.15 shows the result of all our hard work. We have omitted detailed circuits, and just focused on the flow of data and control signals.

Every clock cycle, the processor fetches an instruction from a new PC, fetches the operands, executes the instruction, and writes the results back to the data memory and register file. There are two memory elements in this circuit namely the data and instruction memory. They can possibly refer to the same physical memory structure, or refer to different structures. We shall have ample opportunities to discuss the different schemes in Chapter 11.

The main state elements of the processor are the following registers: *pc*, and $flags$ registers, and the register file. We can optionally add *mar* and *mdr* registers with the memory unit. Note that they are strictly not required in our simple version of the processor. However, they shall be required in advanced designs where a memory request can possibly take multiple cycles. We shall also require them in our microprogrammed processor. Hence, it is a good idea to keep them in our basic design.

Note that most of the sets of wires in the data path have a top-down orientation i.e., the source is above the destination in Figure 9.15. There are two notable exceptions. The source of these wires is below the destination in Figure 9.15. The first such exception is the set of wires that carry the branch target/outcome information from the execute unit to the fetch unit. The second exception is the set of wires that carry the data to be written from the register writeback unit to the register file.

We need to lastly note that the magic of a processor lies in the interplay of the data path and the control path. The control signals give a form to the data path. The unique values of the set of all the control signals determine the nature of instructions. It is possible to change the behavior of instructions, or in fact define new instructions by just changing the control unit. Let us take a deeper look at the control unit.

9.3 The Control Unit

Table 9.4 shows the list of control signals that need to be generated by the control unit along with their associated conditions. The only control signal that is not generated by the control unit is $isBranchTaken$. This is generated by the branch unit that is a part of the execute unit. However, the rest of the 22 signals need to be generated by the control unit. Recall that the inputs to the control unit are the opcode of the instruction, and the value of the *immediate* bit.

The hardwired control unit for our simple processor can be thought of as a black box that takes 6 bits as input (5 opcode bits, and 1 immediate bit), and produces 22 control signals as its output. This is shown in Figure 9.16.

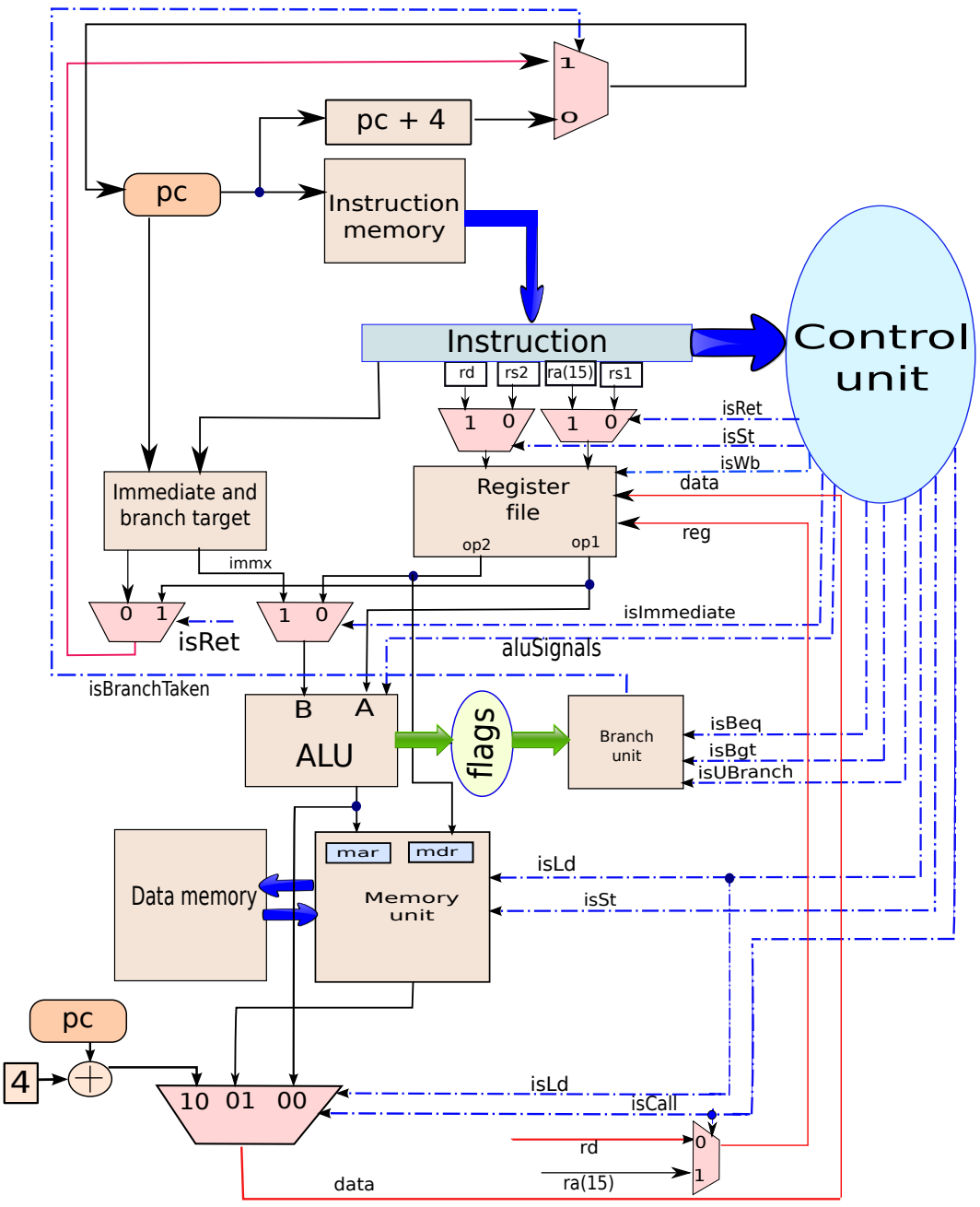

Figure 9.15: A basic processor

Internally, there are a set of logic gates that act on the input bits to produce each output bit. For example, to set the *isAdd* signal, we need to check if the opcode is equal to 00000. Let us number the five bits of the opcode as op_1, op_2, op_3, op_4 and op_5. Here op_1 is the LSB, and op_5 is the MSB. Let us refer to the immediate bit as I.

Serial No.	Signal	Condition
1	$isSt$	Instruction: st
2	$isLd$	Instruction: ld
3	$isBeq$	Instruction: beq
4	$isBgt$	Instruction: bgt
5	$isRet$	Instruction: ret
6	$isImmediate$	I bit set to 1
7	$isWb$	Instructions: add, sub, mul, div, mod, and, or, not, mov, ld, lsl, lsr, asr, $call$
8	$isUBranch$	Instructions: b, $call$, ret
9	$isCall$	Instructions: $call$
$aluSignals$		
10	$isAdd$	Instructions: add, ld, st
11	$isSub$	Instruction: sub
12	$isCmp$	Instruction: cmp
13	$isMul$	Instruction: mul
14	$isDiv$	Instruction: div
15	$isMod$	Instruction: mod
16	$isLsl$	Instruction: lsl
17	$isLsr$	Instruction: lsr
18	$isAsr$	Instruction: asr
19	$isOr$	Instruction: or
20	$isAnd$	Instruction: and
21	$isNot$	Instruction: not
22	$isMov$	Instruction: mov

Table 9.4: List of control signals

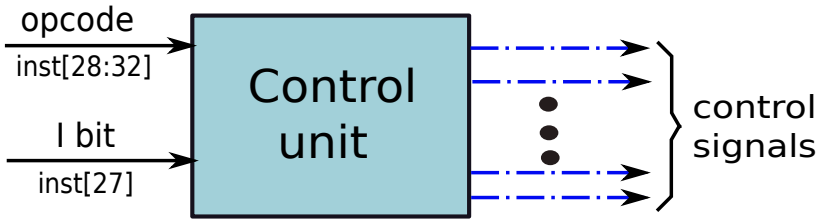

Figure 9.16: Abstraction of a hardwired control unit

Table 9.5 shows the conditions for setting all the control signals. We leave the implementation of Table 9.5 using logic gates as an exercise to the reader. Note that it will take the maximum amount of time to compute the value of $isWb$. Nevertheless, this circuit is extremely simple as compared to a multiplier or a carry lookahead adder. Hence, the total execution time

Serial No.	Signal	Condition
1	$isSt$	$\overline{op_5}.op_4.op_3.op_2.op_1$
2	$isLd$	$\overline{op_5}op_4.op_3.op_2.\overline{op_1}$
3	$isBeq$	$op_5.\overline{op_4}.\overline{op_3}.\overline{op_2}.\overline{op_1}$
4	$isBgt$	$op_5.\overline{op_4}.\overline{op_3}.\overline{op_2}.op_1$
5	$isRet$	$op_5.\overline{op_4}.op_3.\overline{op_2}.\overline{op_1}$
6	$isImmediate$	I
7	$isWb$	$\sim (op_5 + \overline{op_5}.op_3.op_1.(op_4 + \overline{op_2})) + op_5.\overline{op_4}.\overline{op_3}.op_2.op_1$
8	$isUbranch$	$op_5.\overline{op_4}.(\overline{op_3}.op_2 + op_3.\overline{op_2}.\overline{op_1})$
9	$isCall$	$op_5.\overline{op_4}.\overline{op_3}.op_2.op_1$
$aluSignals$		
10	$isAdd$	$\overline{op_5}.\overline{op_4}.\overline{op_3}.\overline{op_2}.\overline{op_1} + \overline{op_5}.op_4.op_3.op_2$
11	$isSub$	$\overline{op_5}.\overline{op_4}.\overline{op_3}.\overline{op_2}.op_1$
12	$isCmp$	$\overline{op_5}.\overline{op_4}.op_3.\overline{op_2}.op_1$
13	$isMul$	$\overline{op_5}.\overline{op_4}.\overline{op_3}.op_2.\overline{op_1}$
14	$isDiv$	$\overline{op_5}.\overline{op_4}.\overline{op_3}.op_2.op_1$
15	$isMod$	$\overline{op_5}.\overline{op_4}.op_3.\overline{op_2}.\overline{op_1}$
16	$isLsl$	$\overline{op_5}.op_4.\overline{op_3}.op_2.\overline{op_1}$
17	$isLsr$	$\overline{op_5}.op_4.\overline{op_3}.op_2.op_1$
18	$isAsr$	$\overline{op_5}.op_4.op_3.\overline{op_2}.\overline{op_1}$
19	$isOr$	$\overline{op_5}.\overline{op_4}.op_3.op_2.op_1$
20	$isAnd$	$\overline{op_5}.\overline{op_4}.op_3.op_2.\overline{op_1}$
21	$isNot$	$\overline{op_5}.op_4.\overline{op_3}.\overline{op_2}.\overline{op_1}$
22	$isMov$	$\overline{op_5}.op_4.\overline{op_3}.\overline{op_2}.op_1$

Table 9.5: Boolean conditions for setting all the control signals

of the control unit is expected to be small as compared to the execute unit.

The hardwired control unit is thus fast and efficient. This is the reason why most commercial processors today use a hardwired control unit. However, hardwired control units are not very flexible. For example, it is not possible to change the behavior of an instruction, or even introduce a new instruction, after the processor has been shipped. Sometimes we need to change the way an instruction is executed if there are bugs in functional units. For example, if the multiplier has a design defect, then it is theoretically possible to run the Booth's multiplication algorithm with the adder, and shift units. We will however, need a very elaborate control unit to dynamically reconfigure the way instructions are executed.

There are other more practical reasons for favoring a flexible control unit. Some instruction sets such as x86 have *rep* instructions that repeat an instruction a given number of times. They also have complicated string instructions that operate on large pieces of data. Supporting such instructions requires a very complicated data path. In principle, we can execute such instructions by having elaborate control units that in turn have simple processors to process these instructions. These sub processors can generate control signals for implementing complicated

CISC instructions.

Way Point 6

1. *We have successfully designed a hardwired processor that implements the entire SimpleRisc ISA.*
2. *Our processor is broadly divided into five stages: IF, OF, EX, MA, and RW.*
3. *The data path contains state elements (such as registers), arithmetic units, logical units, and multiplexers to choose the right set of inputs for each functional unit.*
4. *The multiplexers are controlled by control signals generated by the control unit.*

9.4 Microprogram-Based Processor

Let us now look at a different paradigm for designing processors. We have up till now looked at a processor with a hardwired control unit. We designed a data path with all the elements required to process, and execute an instruction. Where there was a choice between the input operands, we added a multiplexer that was controlled by a signal from the control unit. The control unit took the contents of the instruction as the input, and generated all the control signals. This design style is typically adopted by modern high performance processors. Note that efficiency comes at a cost. The cost is *flexibility*. It is fairly difficult for us to introduce new instructions. We need to possibly add more multiplexers, and generate many more control signals for each new instruction. Secondly, it is not possible to add new instructions to a processor after it has been shipped to the customer. Sometimes, we desire such flexibility.

It is possible to introduce this additional flexibility by introducing a translation table that translates instructions in the ISA to a set of simple microinstructions. Each microinstruction has access to all the latches, and internal state elements of a processor. By executing a group of microinstructions associated with an instruction, we can realize the functionality of that instruction. These microinstructions or *microcodes* are saved in a microcode table. It is typically possible to modify the contents of this table via software, and thus change the way hardware executes instructions. There are several reasons for wanting such kind of flexibility that allows us to add new instructions, or modify the behavior of existing instructions. Some reasons are as follows:

Definition 60

We can have an alternate design style, where we break instructions in the ISA to a set of microinstructions (microcodes). For each instruction, a dedicated unit executes its associated set of microinstructions to implement its functionality. It is typically possible to dynamically change the set of microinstructions associated with an instruction. This helps

us change the functionality of the instruction via software. Such a processor is known as a microprogrammed *processor.*

1. Processors sometimes have bugs in the execution of certain instructions [Sarangi et al., 2006]. This is because of mistakes committed by designers in the design process, or due to manufacturing defects. One such famous example is the bug in division in the Intel® Pentium® processor. Intel had to recall all the Pentium processors that it had sold to customers [Pratt, 1995]. If it had been possible to dynamically change the implementation of the division instruction, then it would not have been necessary to recall all the processors. Hence, we can conclude that some degree of reconfigurability of the processor can help us fix defects that might have been introduced in various stages of the design and manufacturing process.

2. Processors such as Intel Pentium 4, and later processors such as Intel® Core™i3, and Intel® Core™i7 implement some complex instructions by executing a set of microinstructions saved in memory. Complicated operations with strings of data, or instructions that lead to a series of repetitive computations are typically implemented using microcode. This means that the Intel processor internally replaces a complex instruction with a snippet of code containing simpler instructions. This makes it easier for the processor to implement complicated instructions. We do not need to unnecessarily make changes to the data path, add extra state, multiplexers, and control signals to implement complex instructions.

3. Nowadays, processors are part of a chip with many other elements. This is known as a system-on-chip (SOC). For example, a chip in a cellphone might contain a processor, a video controller, an interface to the camera, a sound and network controller. Processor vendors typically hardwire a set of simple instructions, and a lot of other instructions for interfacing with peripherals such as the video and audio controllers are written in microcode. Depending on the application domain and the set of peripheral components, the microcode can be customized.

4. Sometimes custom diagnostic routines are written using a set of dedicated microinstructions. These routines test different parts of the chip during its operation, report faults, and take corrective action. These built-in-self-test (BIST) routines are typically customizable, and are written in microcode. For example, if we desire high reliability, then we can modify the behavior of instructions that perform reliability checks on the CPU to check all components. However, in the interest of time, these routines can be compressed to check fewer components.

We thus observe that there are some compelling reasons to be able to programatically alter the behavior of instructions in a processor to achieve reliability, implement additional features, and improve portability. Hence, modern computing systems, especially, smaller devices such as phones, and tablets use chips that rely on microcode. Such microcode sequences are popularly referred to as *firmware*.

Definition 61
Modern computing systems, especially, smaller devices such as phones, modems, printers, and tablets use chips that rely on microcode. Such microcode sequences are popularly referred to as firmware*.*

Let us thus design a microprogram-based processor that provides us significantly more flexibility in tailoring the instruction set, even after the processor has been fabricated and sent to the customer. Before we proceed to design the data path and control path of a microprogrammed processor, we need to note that there is a fundamental tradeoff between a regular hardwired processor as presented in Section 9.2.7, and a microprogrammed processor. The tradeoff is efficiency versus flexibility. We cannot expect to have a very flexible processor that is fast and power efficient. Let us keep this important tenet in mind and proceed with the design.

9.5 Microprogrammed Data Path

Let us design the data path for a microprogrammed processor. Let us not design it from scratch. Let us rather modify the data path for the processor as shown in Section 9.2.7. Recall that it had some major units such as the fetch unit, register file, ALU, branch unit, and memory unit. These units were connected with wires, and whenever there was a possibility of multiple source operands, we added a multiplexer in the data path. The role of the control unit was to generate all the control signals for the multiplexers.

The issue is that the connections to the multiplexers are hardwired. It is not possible to establish arbitrary connections. For example, it is not possible to send the output of the memory unit to the input of the execute unit. Hence, we wish to have a design that is free of fixed interconnections between components. It should be theoretically possible for any unit to send data to any other unit.

The most flexible interconnect is a bus-based structure. A *bus* is a set of common copper wires that connect all the units. It supports one writer, and multiple readers at any point of time. For example, unit A can write to the bus at a certain point of time, and all the other units can get the value that unit A writes. It is possible to send data from one unit to another unit, or from one unit to a set of other units if required. The control unit needs to ensure that at any point of time, only one unit writes to the bus, and the unit that needs to process the value that is being written, reads the value from the bus.

Definition 62 *A* bus *is a set of common wires that connect all the functional units in a processor. It supports one writer, and multiple readers at any point of time.*

Let us now proceed to design simplified versions of all the units that we introduced for our hardwired processor. These simplified versions can aptly be used in the data path of our microprogrammed processor.

9.5.1 Fetch Unit

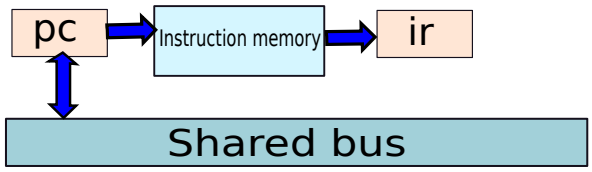

Figure 9.17: The fetch unit in a microprogrammed processor

Let us start out by explaining the design philosophy of the microprogrammed processor. We add registers with every unit. These registers store the input data for the specific unit, and a dedicated output register stores the results generated by the unit. Both these sets of registers are connected to the common bus. Unlike the hardwired processor, where there was a good amount of coupling across different units, units in a microprogrammed processor are fairly independent of each other. Their/ job is to perform a set of actions, and put the results back on to the bus. Each unit is like a function in a programming language. It has an interface consisting of a set of registers to read data in. It typically takes 1 cycle to compute its output, and then the unit writes the output value to an output register.

In concordance with this philosophy, we present the design of the fetch unit in Figure 9.17. It has two registers – *pc* (PC), and *ir* (instruction register). We shall use the acronym, IR, for the instruction register. *ir* contains the contents of the instruction. The *pc* register can read its value from the bus, and can also write its value to the bus. We have not connected *ir* to the bus because no other unit is typically interested in the exact contents of the instruction. Other units are only interested in different fields of the instruction. Hence, it is necessary to decode the instruction and break it into a set of different fields. This is done by the decode unit.

9.5.2 Decode Unit

This unit is similar in function to the operand fetch unit as described in Section 9.2.3. However, we do not include the register file in this unit. We treat it as a separate unit in the microprogrammed processor. Figure 9.18 shows the design of the operand fetch unit.

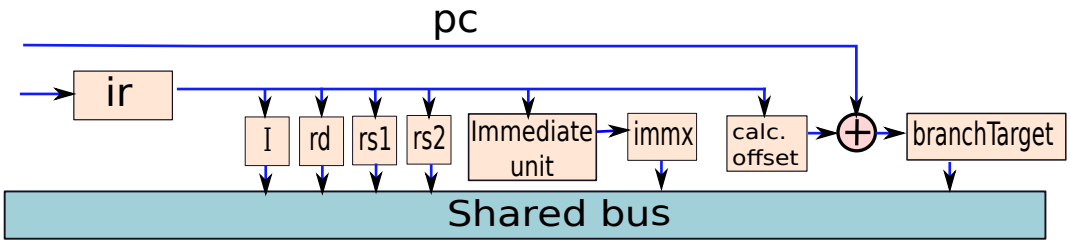

Figure 9.18: The decode unit in a microprogrammed processor

The job of the decode unit is to break down the contents of the instruction into multiple

fields, and export them as registers. In specific, the following registers are made available to the bus, I (immediate bit), rd (destination register), $rs1$ (source register 1), $rs2$ (source register 2), $immx$ (after processing the modifiers), and $branchTarget$ (branch target). To compute the branch target we calculate the offset from the current PC by extracting bits [1:27] from ir, and shifting it to the left by 2 places. This is added to the current value of the PC. Table 9.6 shows the range of bits extracted from ir for each output register.

Register	Bits in ir
I	27
rd	23-26
$rs1$	19-22
$rs2$	15-18
$immx$	1-18 (process modifiers)
$branchTarget$	PC + $(ir[1:27] \ll 2)$

Table 9.6: List of bits in ir corresponding to each register in the decode unit

It is possible that a given program execution might not have values for all the registers. For example, an instruction in the *register* format will not have an embedded immediate value. Hence, in this case the $immx$ register will have junk data. However, it does not hurt to extract all possible fields from the instruction, and store them in registers. We can use only those registers that contain valid data, and ignore those registers that are not relevant to the instruction. This ensures that our data path remains simple, and we do not need costly multiplexers in the decode unit.

9.5.3 Register File

We had combined the decode unit and the register file, into one unit called the operand fetch unit of the hardwired processor. However, we prefer to keep the register file separate in the microprogrammed processor. This is because in the hardwired processor it was accessed right after decoding the instruction. However, this might not be the case in the microprogrammed processor. It might need to be accessed possibly several times during the execution of an instruction.

The register file has two source registers – $regSrc$, and $regData$. The $regSrc$ register contains the number of the register that needs to be accessed. In the case of a write operation, the $regData$ register contains the value to be written. The $args$ values are directly read from the bus. They contain the commands to the register file. We assume that there are dedicated wires in the shared bus to carry the arguments ($args$) values. They take different values, where each value corresponds to a unique operation of a unit. The value 00...0 is a distinguished value that corresponds to a *nop* (no operation).

The arguments to the register file, are very simple – *read*, *write*, and *nop*. If the args specify a *write* operation, then the value in $regData$ is written to the register specified by the $regSrc$ register. If a *read* operation is specified, then the register specified by $regSrc$ is read and its value is stored in the register, $regVal$ (register value).

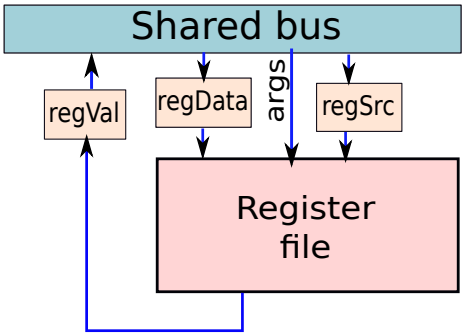

Figure 9.19: The register file in a microprogrammed processor

To access the register file it is thus necessary to write the number of the register to the $regSrc$ register, write the value to be written to the $regData$ register if required, and finally specify the appropriate arguments. The assumption is that after 1 cycle the operation is complete. In the case of a read operation, the value is available in the $regVal$ register.

9.5.4 ALU

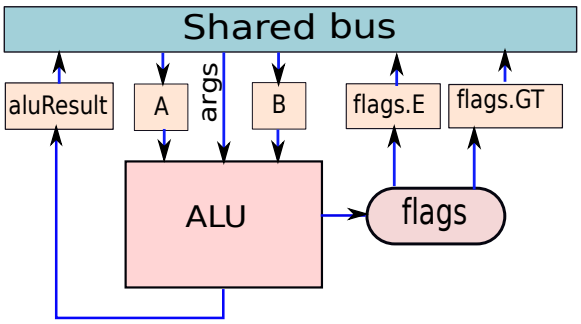

Figure 9.20: The ALU in a microprogrammed processor

The structure of the ALU is shown in Figure 9.20. It has two input registers, A and B. The ALU performs actions on the values contained in registers, A and B. The nature of the operation is specified by the $args$ value. For example, if it specifies an add operation, then the ALU adds the values contained in registers, A and B. If it specifies a subtract operation, then the value in B is subtracted from the value contained in A. For the *cmp* instruction, the ALU updates the flags. Recall that in *SimpleRisc* we use two flags that specify the equality, and greater than conditions. They are saved in the registers $flags.E$ and $flags.GT$, respectively. The result of the ALU operation is then saved in the register $aluResult$. Here also, we assume that the ALU takes 1 cycle to execute after the args values are specified on the bus.

9.5.5 Memory Unit

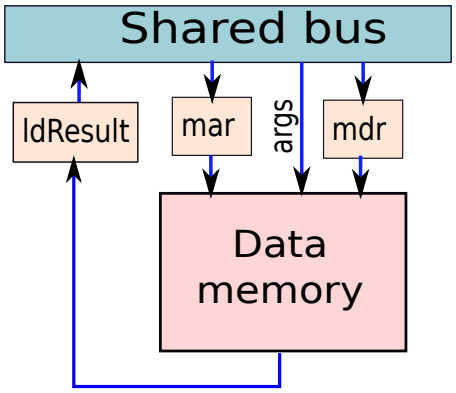

Figure 9.21: The memory unit in a microprogrammed processor

The memory unit is shown in Figure 9.21. Like the hardwired processor, it has two source registers – *mar* and *mdr*. The memory address register(*mar*) buffers the memory address, and the memory data register (*mdr*) buffers the value that needs to be stored. Here also, we require a set of arguments that specify the nature of the memory operation – load or store. Once, a load operation is done, the data is available in the *ldResult* register.

9.5.6 Overview of the Data Path

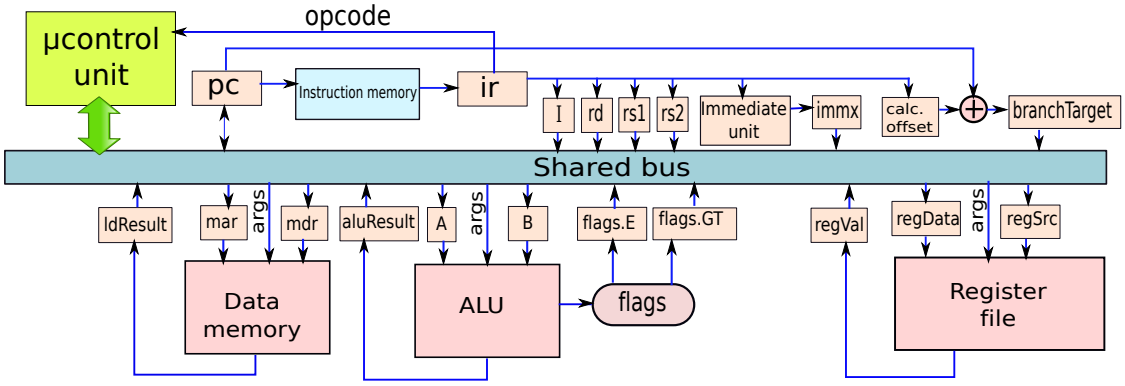

Figure 9.22: The data path in a microprogrammed processor

Let us now add all the individual units, and take a look at the entire data path as shown in Figure 9.22. Along with all the units that we just described, we have added an extra unit, which is the microprogrammed control unit (μcontrol unit). Its role is to execute a set of microinstructions corresponding to each program instruction, and orchestrate the flow of data

values across the different units in the data path of the microprogrammed processor. It is mainly responsible for the execution of microinstructions, data transfers across the different units, and for transferring control to the correct program instruction by updating the PC. Note that we have also added an extra connection between the *ir* register and the μcontrol unit to transfer the opcode of the instruction. We require the μcontrol unit to load the appropriate set of microinstructions corresponding to the program instruction. By design, we do not wish to make the value of the opcode available to other units. This is because, we have a set of microinstructions for each opcode, and there is no reason why other units should require the value of the opcode.

Definition 63

The microcontrol unit*, also referred to as the μcontrol unit is a dedicated piece of hardware that is responsible for the execution of a set of microinstructions corresponding to each program instruction. Its role is to fetch the appropriate set of microinstructions from a dedicated* microprogram memory*, and execute them in sequence. A register called the* micro PC *(μpc) points to the currently executing microinstruction.*

We envision a microprogram memory that is a part of the μcontrol unit. It contains the set of microinstructions corresponding to each program instruction. It is thus necessary for the μcontrol unit to jump to the starting address of the set of microinstructions corresponding to each program instruction. We also need a microPC that points to the current microinstruction being executed.

Before discussing the design and implementation of the μcontrol unit, let us first look at programming, or rather microprogramming our new processor. We need to design a microassembly language that will help us write programs for it.

9.6 Microassembly Language

9.6.1 Machine Model

All the internal registers in Figure 9.22 are the set of registers that are visible to microassembly instructions. Ideally microassembly instructions are not supposed to be aware of regular architectural registers, and other aspects of architectural state. They are only supposed to be aware of internal registers that are not externally visible.

Table 9.7 shows the list of internal registers in our microprogrammed data path. Note that we have 1-bit registers, 4-bit registers, and 32-bit registers.

Microprogrammed instructions do not access memory. Hence, they do not need a view of memory.

9.6.2 Microinstructions

Let us look at the life cycle of a regular program instruction. The first step is to fetch the contents of the instruction from the instruction memory. Let us introduce a microinstruction

Serial No.	Register	Size (bits)	Function
1	pc	32	program counter
2	ir	32	instruction register
3	I	1	immediate bit in the instruction
4	rd	4	destination register id
5	$rs1$	4	id of the first source register
6	$rs2$	4	id of the second source register
7	$immx$	32	immediate embedded in the instruction (after processing modifiers)
8	$branchTarget$	32	branch target, computed as the sum of the PC and the offset embedded in the instruction
9	$regSrc$	4	contains the id of the register that needs to be accessed in the register file
10	$regData$	32	contains the data to be written into the register file
11	$regVal$	32	value read from the register file
12	A	32	first operand of the ALU
13	B	32	second operand of the ALU
14	$flags.E$	1	the equality flag
15	$flags.GT$	1	the greater than flag
16	$aluResult$	32	the ALU result
17	mar	32	memory address register
18	mdr	32	memory data register
19	$ldResult$	32	the value loaded from memory

Table 9.7: List of all the internal registers

to read the contents of the instruction from the instruction memory and place it in the IR (ir). Let us call it $mloadIR$. Note that we will add the prefix m (m for micro) to every microinstruction. This is to denote the fact that it is a microinstruction, and differentiate it from regular program instructions.

Microinstruction	Semantics
$mloadIR$	Loads the ir with the contents of the instruction

Once, we have loaded the instruction register, it automatically sends the contents to all the subunits in the decode unit, and they extract the appropriate bit fields, and save them in the decode registers – I, rd, $rs1$, $rs2$, $immx$, and $branchTarget$. We envision an *mdecode* instruction in the 0-address format that makes the μcontrol unit wait for 1 cycle. In this cycle, all the decode registers get populated.

Microinstruction	Semantics
mdecode	Waits for 1 cycle. Meanwhile, all the decode registers get populated.

Note that these two steps (*mloadIR* and *mdecode*) are common for all program instructions. After this, we need to load the microinstructions for the specific program instruction. This is achieved through an *mswitch* instruction that instructs the μcontrol unit to jump to the appropriate location in the microinstruction memory, and begins executing microinstructions starting from that location.

Microinstruction	Semantics
mswitch	Load the set of microinstructions corresponding to the program instruction

Now, the processing of the instruction can start. The aim here is to use as few microinstructions as possible. We want to keep the microassembly interface very simple. Let us first introduce the *mmov* instruction that moves data from the source register to a destination register. Additionally, it can set the arguments of the unit corresponding to the destination register. We thus introduce a 2-address and 3-address format of the *mmov* instruction. The 3-address format contains the arguments ($args$) of the unit corresponding to the destination register, as shown below.

Microinstruction	Semantics
mmov $r1$, $r2$	$r1 \leftarrow r2$
mmov $r1$, $r2$, $\langle args \rangle$	$r1 \leftarrow r2$, send the value of $args$ on the bus

We sometimes face the need to load constants into registers. Hence, we introduce the *mmovi* instruction that loads a constant into a register.

Microinstruction	Semantics
mmovi $r1$, $\langle imm \rangle$	$r1 \leftarrow imm$
mmovi $r1$, $\langle imm \rangle$, $\langle args \rangle$	$r1 \leftarrow imm$, send the value of $args$ on the bus

We need an *madd* instruction because we need to increment the values of registers such as the *pc*. Instead of using the main ALU, we can have a small adder as a part of the μcontrol unit. We refer to this as the μadder. Here, there is a tradeoff to make. Do we need an add instruction that adds two registers, and saves it in another register? At the microinstruction

level, this is seldom required. We definitely do not require this instruction to implement the *SimpleRisc* instruction set. Hence, we do not see a reason to include this microinstruction. If there is ever a need to have one such microinstruction, then we can always use the main ALU in the data path to perform the addition. We thus introduce a simple add instruction in the 2-address format. It adds an immediate value to a register. The semantics of this instruction is shown below.

Microinstruction	Semantics
madd $r1$, $\langle imm \rangle$	$r1 \leftarrow r1 + imm$
madd $r1$, $\langle imm \rangle$, $\langle args \rangle$	$r1 \leftarrow r1 + imm$, send the value of $\langle args \rangle$ on the bus

Here, the *madd* instruction adds imm to $r1$, and saves the result in $r1$. imm can be a positive or a negative number. We restrict it to a 12-bit number, because we do not need more bits in most cases. The range of the immediate is thus between -2048 and +2047.

Lastly, we need branch instructions. We need both conditional branches, and unconditional branches. We thus introduce two new microinstructions – *mb* (branch) and *mbeq* (branch if the arguments are equal). The *mb* microinstruction takes a single argument, which is the address of the target microinstruction (or its label while writing microassembly code). We use the PC-direct addressing mode here as compared to the PC-relative addressing mode because, we expect the total number of microinstructions to be small. Secondly, if we would have use a PC-relative addressing mode, then we would have required an extra adder in our data path to add the offset to the PC. The *SimpleRisc* instruction set allocates 5 bits for the opcode. This means that at the most we can have 32 instructions in our instruction set. Let us assume that in the worst case, an instruction translates to 20 microinstructions. We would thus need to store 640 microinstructions. We can thus allocate 10 bits for the specifying the address of the microinstruction and our μpc (micro-PC) can also be 10 bits wide. This means that at the most we can support a total of 1024 microinstructions. This is much more than what we actually require. However, it is not a bad idea to over design hardware because it cannot be changed later. Note that in the microinstruction memory, the address refers to the index of the microinstruction (not to the starting address of the first byte).

The *mbeq* instruction requires three operands. The first operand is a register, the second operand is an immediate, and the third operand is the address(label) of a microinstruction. If the value contained in the register is equal to the immediate operand, then the microPC jumps to the microinstruction specified by the third operand. Otherwise, the next microinstruction in sequence is executed.

Microinstruction	Semantics
mb $\langle addr \rangle$	execute the microinstruction at $\langle addr \rangle$(label) in the microprogram memory
mbeq reg, imm, $\langle addr \rangle$	If the value in the internal register reg is equal to imm, then the microPC needs to jump to $\langle add \rangle$(label)

Serial No.	Microinstruction	Semantics
1	$mloadIR$	$ir \leftarrow [pc]$
2	$mdecode$	populate all the decode registers
3	$mswitch$	jump to the μpc corresponding to the opcode
4	$mmov\ reg1, reg2, \langle args \rangle$	$reg1 \leftarrow reg2$, send the value of $args$ to the unit that owns $reg1$, $\langle args \rangle$ is optional
5	$mmovi\ reg1, imm, \langle args \rangle$	$reg1 \leftarrow imm$, $\langle args \rangle$ is optional
6	$madd\ reg1, imm, \langle args \rangle$	$reg1 \leftarrow reg1 + imm$, $\langle args \rangle$ is optional
7	$mbeq\ reg1, imm, \langle addr \rangle$	if $(reg1 = imm)$ $\mu pc \leftarrow addr(label)$
8	$mb\ \langle addr \rangle$	$\mu pc \leftarrow addr(label)$

Table 9.8: List of microinstructions

To summarize, Table 9.8 shows the 8 microinstructions that we have described in this section. We have a compact list of 8 microinstructions, and thus we can encode each microinstruction using just 3 bits.

9.6.3 Implementing Instructions in the Microassembly Language

Let us now try to implement program instructions in the microassembly language using the set of basic microinstructions enumerated in Table 9.8.

For all the instructions, they start with a common set of microinstructions as shown below. We refer to these 4 microinstructions as the *preamble*.

```
1 .begin:
2 mloadIR
3 mdecode
4 madd pc, 4
5 mswitch
```

Definition 64
A set of microinstructions that is common to all program instructions and is executed at the beginning before proceeding to implement the logic of the instruction, is known as the preamble*, or* microcode preamble*.*

Every instruction needs to pass through at least three of these steps. We need to fetch the contents of the PC and load them into the *ir* register. Then, we need to decode the instruction, and break it down into its constituent fields. For instructions, other than branches, we need to increment the value of the PC by 4. In our microcode we prefer to do this step for all the instructions. For taken branches, we need to later overwrite the PC with the branch target. Lastly, we need to execute the *mswitch* instruction to jump to the starting location of the set of microinstructions that are specific to the program instruction.

The label *.begin* points to the beginning of this routine. Note that after finishing the execution of an instruction, we need to jump to the *.begin* label such that we can start processing the next instruction in the program. Note that in our microassembly code we specify the label that we need to branch to. When the microassembly code is translated to actual machine level microinstructions, then each label is replaced by the address of the corresponding microinstruction.

9.6.4 3-Address Format ALU Instructions

Let us now look at implementing 3-address format ALU instructions. These instructions are: *add*, *sub*, *mul*, *div*, *mod*, *and*, *or*, *lsl*, *lsr*, and *asr*.

First, we need to read the value of the first operand stored in $rs1$ from the register file, and send it to the ALU. The microcode snippet to achieve this is as follows:

```
1 mmov regSrc, rs1, <read>
2 mmov A, regVal
```

Note, that we are combining a functional unit operation, and a register transfer in the same cycle. This can be confusing at the beginning. Hence, the reader should read this example several times and ensure that she has a clear understanding. The reason that we fuse both the operations is because microcode registers are typically very small, and thus they can be accessed very quickly. Hence, it is not a good idea to use a complete cycle for transferring data between micro registers. It is a better idea to fuse a register transfer with a functional unit operation, such that we can ensure that we are roughly doing a similar amount of work every cycle.

Let us proceed. Subsequently, we need to check if the second operand is a register or an immediate. This can be achieved by comparing the I register with 1. If it is 1, then the second operand is an immediate, else it is a register. The following piece of code first checks this condition, and then performs data transfers accordingly.

```
1 mbeq I, 1, .imm
2 /* second operand is a register */
3 mmov regSrc, rs2, <read>
4 mmov B, regVal, <aluop>
5 mb .rw
6 /* second operand is an immediate */
7 .imm:
8 mmov B, immx, <aluop>
```

```
/* write the ALU result to the register file*/
.rw:
```

Here, we first check if the value stored in the I register is equal to 1, using the *mbeq* instruction. If it is not 1, then the second operand is a register, and we start executing the subsequent microinstruction. We move the contents of the register, $rs2$, to the $regSrc$ register that contains the index of the register that we need to read from the register file. Then we move the value of the operand read from the register file ($regVal$) to the ALU (register B). Since the value in register A is already present, we can directly start the ALU operation. This is indicated to the ALU by sending an extra argument ($\langle aluop \rangle$) that encodes the ALU operation. $\langle aluop \rangle$ corresponds to one of the following operations: *add*, *sub*, *mul*, *div*, *mod*, *and*, *or*, *lsl*, *lsr*, and *asr*.

However, if the value of the I register is 1, then we need to branch to *.imm*. The value of the immediate embedded in the instruction is already available with appropriate sign extensions in the register $immx$. We need to simply transfer the value of $immx$ to B (second ALU register), and the arguments ($\langle aluop \rangle$) to the ALU. Similar to the case with the second operand being a register, $\langle aluop \rangle$ encodes the ALU operation. Once, we are done, we need to start execution at the label, *.rw*.

The label *.rw* needs to point to code that writes the value of the computed result to the register file, and then proceeds to execute the next instruction. The code for these two operations is shown below.

```
.rw:
        mmov regSrc, rd
        mmov regData, aluResult, <write>
        mb .begin
```

We write the result of the ALU into the register file, and then branch to the beginning, where we proceed to execute the next instruction. To summarize, here is the code for any 3-address format ALU instruction (other than the preamble).

```
/* transfer the first operand to the ALU */
mmov regSrc, rs1, <read>
mmov A, regVal

/* check the value of the immediate register */
mbeq I, 1, .imm
/* second operand is a register */
mmov regSrc, rs2, <read>
mmov B, regVal, <aluop>
mb .rw
/* second operand is an immediate */
.imm:
mmov B, immx, <aluop>

```

```
15 /* write the ALU result to the register file*/
16 .rw:
17 mmov regSrc, rd
18 mmov regData, aluResult, <write>
19 mb .begin
```

This code snippet has 10 microinstructions. Recall that we also need to execute 4 more microinstructions as a part of the preamble before this. They read the PC, decode the instruction, set the next PC, and jump to the beginning of the appropriate set of microinstructions. Executing 14 microinstructions for 1 program instruction is clearly a lot of effort. However, the reader must recall that we are not really after performance here. We wanted to design a very clean and flexible means of accessing different units.

9.6.5 2-Address Format ALU Instructions

The three ALU instructions in the 2-address format, are *not*, *mov*, and *cmp*. *not* and *mov* have a similar format. They do not use the first source operand, $rs1$. They operate on either $rs2$, or $immx$, and transfer the result to the register pointed by rd.

Let us look at the *mov* instruction first. We first check whether the second operand is an immediate, or not, by comparing the value in register I with 1. If it is equal to 1, then we jump to the label, *.imm*. Otherwise, we proceed to execute the subsequent instructions in Lines 4, and 5. In Line 4, we transfer $rs2$ to $regSrc$, along with the $\langle read \rangle$ command. The operand is read and stored in $regVal$. In the next cycle, we transfer $regVal$ to $regData$ such that it can be written back to the register file. If the second operand was an immediate, then we execute the code in Line 9. We transfer the immediate (stored in $immx$) to the $regData$ register. In either case, $regData$ contains the value to be written to the register file. Then we transfer the id of the destination register (stored in rd) to $regSrc$, and simultaneously issue the write command in Line 13.

mov instruction

```
1  /* check the value of the immediate register */
2  mbeq I, 1, .imm
3  /* second operand is a register */
4  mmov regSrc, rs2, <read>
5  mmov regData, regVal
6  mb .rw
7  /* second operand is an immediate */
8  .imm:
9  mmov regData, immx
10
11 /* write to the register file*/
12 .rw:
13 mmov regSrc, rd, <write>
14
15 /* jump to the beginning */
16 mb .begin
```

Let us now write a similar routing for the *not* instruction. The only additional step is to transfer the value read from the register to the ALU, compute the logical negation, and then transfer the value back to the register file. A hallmark feature of our microassembly language is that we can transfer a value to a unit, and if all the other operands are in place, then we can also perform an operation in the unit in the same cycle. The implicit assumption here is that 1 clock cycle is enough to transfer the data between registers, and perform a computation. In line with this philosophy we transfer the value of $immx$, or $regVal$ to register B of the ALU, and also perform a $\langle not \rangle$ operation in the same cycle (see Lines 5 and 9). Like the *mov* instruction, we transfer the ALU result to the $regData$ register, and write it to the register file in Lines 13, and 14.

not instruction

```
1  /* check the value of the immediate register */
2  mbeq I, 1, .imm
3  /* second operand is a register */
4  mmov regSrc, rs2, <read>
5  mmov B, regVal, <not> /* ALU operation */
6  mb .rw
7  /* second operand is an immediate */
8  .imm:
9  mmov B, immx, <not> /* ALU operation */
10
11 /* write to the register file*/
12 .rw:
13 mmov regData, aluResult
14 mmov regSrc, rd, <write>
15
16 /* jump to the beginning */
17 mb .begin
```

Let us now look at the compare instruction that does not have a destination operand. It compares two operands, where one is a register operand, and the other can be either a register or an immediate. It saves the results automatically in the $flags.E$ and $flags.GT$ registers.

Let us now consider the microcode for the *cmp* instruction.

cmp instruction

```
1  /* transfer rs1 to register A */
2  mmov regSrc, rs1, <read>
3  mmov A, regVal
4
5  /* check the value of the immediate register */
6  mbeq I, 1, .imm
7  /* second operand is a register */
8  mmov regSrc, rs2, <read>
9  mmov B, regVal, <cmp> /* ALU operation */
10 mb .begin
11
```

```
12 /* second operand is an immediate */
13 .imm:
14 mmov B, immx, <cmp> /* ALU operation */
15 mb .begin
```

Here, we first transfer the value in register $rs1$ to the ALU (in register A). Then, we check if the second operand is an immediate. If it is an immediate, then we transfer the value of $immx$ to the ALU (in register B), and simultaneously issue a command to execute a compare in Line 14. However, if the second operand is a register, then we need to read it from the register file (Line 8), and then transfer it to the ALU (Line 9). The last step is to branch to the beginning (*mb .begin*).

9.6.6 The *nop* Instruction

Implementing the *nop* instruction is trivial. We just need to branch to the beginning as shown below.

```
1 mb .begin
```

9.6.7 *ld* and *st* instructions

Let us now look at the load instruction. We need to transfer the value of the first source register to the ALU. Then we transfer the value of the immediate to the second ALU register (B), and initiate the add operation to calculate the effective address. Once the effective address has been calculated, it is available in the *aluResult* register. Subsequently, we move the contents of the *aluResult* register to the memory address register (*mar*), and initiate a load operation. The result is available in the *ldResult* register in the next cycle. We write the loaded value to the register specified by rd in the next two cycles.

ld instruction

```
1  /* transfer rs1 to register A */
2  mmov regSrc, rs1, <read>
3  mmov A, regVal
4
5  /* calculate the effective address */
6  mmov B, immx, <add> /* ALU operation */
7
8  /* perform the load */
9  mmov mar, aluResult, <load>
10
11 /* write the loaded value to the register file */
12 mmov regData, ldResult
13 mmov regSrc, rd, <write>
14
15 /* jump to the beginning */
16 mb .begin
```

The microcode for the store instruction is similar to that of the load instruction. We first calculate the effective memory address and store it in the *mar* register. Then we read the value of the *rd* register that contains the data to be stored (Line 10). We save this in the *mdr* register, and issue the store (Line 11).

st instruction

```
1  /* transfer rs1 to register A */
2  mmov regSrc, rs1, <read>
3  mmov A, regVal
4
5  /* calculate the effective address */
6  mmov B, immx, <add> /* ALU operation */
7
8  /* perform the store */
9  mmov mar, aluResult
10 mmov regSrc, rd, <read>
11 mmov mdr, regVal, <store>
12
13 /* jump to the beginning */
14 mb .begin
```

9.6.8 Branch Instructions

There are five branch instructions in *SimpleRisc* : *b*, *beq*, *bgt*, *call*, and *ret*.

Implementing the unconditional branch instruction is trivial. We simply need to transfer the value of the branch target to the PC.

b instruction

```
1  mmov pc, branchTarget
2  mb .begin
```

We can make a minor modification to this code to implement the *beq*, and *bgt* instructions. We need to check the value of the flags registers, and set the branchTarget to the PC only if the corresponding flags register contains a 1.

beq instruction

```
1  /* test the flags register */
2  mbeq  flags.E, 1, .branch
3  mb. begin
4
5  .branch:
6  mmov pc, branchTarget
7  mb .begin
```

bgt instruction

```
1  /* test the flags register */
2  mbeq  flags.GT, 1, .branch
3  mb. begin
4
5  .branch:
6  mmov pc, branchTarget
7  mb .begin
```

The last two instructions that we need to implement are the *call* and *ret* instructions. The *call* instruction is a combination of a simple branch, and a register write operation that adds the value of the next PC ($PC + 4$) to the return address register (register 15). The microcode

is as follows. Note that we do not increment the PC by 4 because it is already incremented in the preamble.

call instruction

```
1 /* save PC + 4 in the return address register */
2 mmov regData, pc
3 mmovi regSrc, 15, <write>
4
5 /* branch to the function */
6 mmov pc, branchTarget
7 mb .begin
```

We save the address of the next PC in the register file in lines 2 to 3. Then we move the *branchTarget* to the PC, and then proceed to execute the first instruction in the invoked function.

The *ret* instruction performs the reverse operation, and transfers the return address to the PC.

ret instruction

```
1
2 /* save the contents of the return
3                 address register in the PC */
4 mmovi regSrc, 15, <read>
5 mmov pc, regVal
6 mb .begin
```

We have thus implemented all our *SimpleRisc* instructions in microcode. A microcoded implementation is definitely slower that our hardwired data path. However, we have gained a lot in terms of flexibility. We can implement some very complex instructions in hardware, and thus make the task of software developers significantly easier. We can also dynamically change the behavior of instructions. For example, if we wish to store the return address on the stack rather than the return address register, we can do so easily (see Examples 132 and 133).

Example 132

Change the call instruction to store the return address on the stack. The preamble need not be shown (study carefully).

Answer:

stack-based call instruction

```
1
2 /* read and update the stack pointer */
3 mmovi regSrc, 14, <read> /* regSrc contains the id
4                   of the stack pointer */
5 madd regVal, -4 /* decrement the stack pointer */
6 mmov mar, regVal /* MAR contains the new stack pointer */
7
```

```
mmov regData, regVal, <write> /* update the stack pointer */

/* write the return address to the stack */
mmov mdr, pc, <store>

mb. begin
```

Example 133

Change the ret instruction to load the return address from the stack. The preamble need not be shown.

Answer:

stack-based call instruction

```
/* read the stack pointer */
mmovi regSrc, 14, <read>

/* set the memory address to the stack pointer */
mmov mar, regVal, <load>

mmov pc, ldResult /* set the PC */

/* update the stack pointer */
madd regVal, 4 /* sp = sp + 4 */
mmov regData, regVal, <write> /* update stack pointer */

/* jump to the beginning */
mb .begin
```

Example 134

Implement an instruction to compute the factorial of the number saved in register r2. You can destroy the contents of r2. Save the result in register r3. Assume that the number is greater than 1.

stack-based call instruction

```

/* code to set the inputs to the multiplier */
mmovi B, 1
mmovi regSrc, 2, <read>
mmov  A, regVal
/* at this point A = r2, B = 1 */
```

```
7
8  /* loop */
9  .loop:
10 /* Now begin the multiplication */
11 mmov B, B, <multiply> /* aluResult = A * B */
12 mmov B, aluResult /* B = aluResult */
13
14 /* decrement and test */
15 madd A, -1 /* A = A - 1 */
16 mbeq A, 1, .out /* compare A with 1 */
17 mb .loop
18
19 .out:
20 mmov regData, aluResult
21 mmovi regSrc, 3, <write> /* all done */
22
23 mb .begin
```

Example 135

Implement an instruction to find if the value saved in register $r2$ is a cubic Armstrong Number. A cubic Armstrong number is equal to the sum of the cubes of its decimal digits. For example, 153 is one such number. $153 = 1^3 + 5^3 + 3^3$. Save the Boolean result in $r3$. Assume two scratch registers: $sr1$ and $sr2$.

stack-based call instruction

```
1
2  /* Set the inputs of the ALU */
3  mmovi regSrc, 2, <read>
4  mmov  A, regVal
5  mmov  sr1, regVal
6  mmovi B, 10
7  mmovi sr2, 0 /* sum = 0 */
8
9  /* loop */
10 .loop:
11 /* test */
12 mbeq A, 0, .out
13
14 /* compute the mod and cube it */
15 mmov B, B, <mod> /* aluResult = A % B */
16 mmov B, aluResult /* B = aluResult */
17 mmov A, aluResult, <multiply> /* aluResult = (A%B)^2 */
```

```
18 mmov A, aluResult, <multiply> /* aluResult = (A%B)^3 */
19 mmov A, aluResult /* A =  (A%B)^3 */
20 mmov B, sr2, <add> /* add the running sum */
21 mmov sr2, aluResult /* sr2 has the new sum */
22
23 /* test */
24 mmov A, sr1 /* old value of A */
25 mmovi B, 10, <divide>
26 mmov A, aluResult /* A = A / 10 */
27 mmov sr1, A /* sr1 = A */
28
29 mb .loop
30
31 /* out of the loop */
32 .out:
33 mmov A, sr2 /* A contains the sum */
34 mmov B, regVal, <cmp> /* compare */
35 mmov regSrc, 3
36 mbeq flags.E, 1, .success
37
38 /* failure */
39 mmov regData, 0, <write>
40 mb .begin
41
42 .success:
43 mmov regData, 1, <write>
44 mb .begin
```

Example 136

Implement an instruction to test if a number saved in register r2 is prime. Assume that the number is greater than 3. Save the result in r3.

stack-based call instruction

```
1
2 /* Read the register and set the ALU inputs */
3 mmovi regSrc, 2, <read>
4 mmov  A, regVal
5 mmovi B, 1
6
7 .loop:
8         /* test for divisibility */
9         madd B, 1, <mod> /* aluResult = A % (B+1), B = B + 1 */
```

```
10          mbeq aluResult, 0, .failure
11
12          /* test B */
13          mmov A, A, <cmp> /* compare A with B */
14          mbeq flags.E, 1, .success
15
16  mb .loop
17
18  .success:
19          mmovi regSrc, 3
20          mmovi regData, 1, <write>
21          mb .begin
22
23  .failure:
24          mmovi regSrc, 3
25          mmovi regData, 0, <write>
26          mb. begin
```

9.7 Shared Bus and Control Signals

Let us take a look again at the list of implemented microinstructions in Table 9.8. We observe that each microinstruction has at the most one register read operand, and one register write operand. We typically read from one internal register, and then use it as a part of a computation (addition or comparison), and then write the result to another internal register.

We thus propose the design of a shared bus that actually consists of two buses as shown in Figure 9.23. The first bus is known as the write bus that is connected to all the registers that might potentially write data to the bus. The output of the write bus, the embedded immediate (μimm) in the microinstruction, and the output of the μadder are sent to a multiplexer. Recall that the μadder adds the embedded immediate with the contents of a register. Now, this multiplexer chooses one value among the three, and then sends it on the read bus. We refer to this multiplexer as the *transfer multiplexer*. All the registers that might potentially read a value are connected to the read bus. The PC is connected to both the buses. The μadder has two inputs. One of them is the sign extended immediate that is a part of the microinstruction, and the other is the output of the write bus.

Simultaneously, we compare the value sent on the write bus with the embedded immediate (μimm). The result is contained in the $isMBranch$ signal. The $isMBranch$ signal is required for implementing the *mbeq* instruction.

To create a flexible data path, we need to add as many interconnections between units as possible. We thus decide to connect every register other than the decode, and flags registers to both the read and write buses. These registers are the input/output registers of the register file ($regSrc$, $regData$, and $regVal$), the ALU registers (A, B, $aluResult$), and the registers

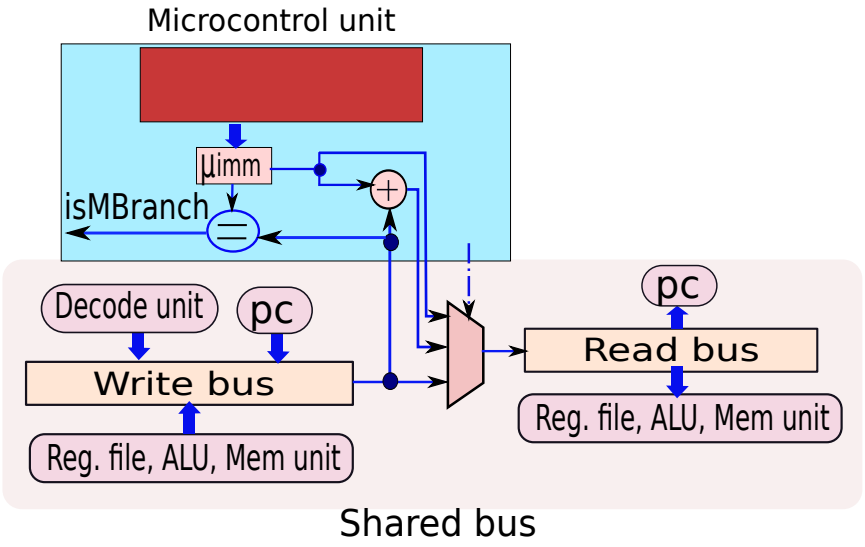

Figure 9.23: The design of the shared bus

associated with the memory unit (mar, mdr, $ldResult$). To support branch instructions, it is also necessary to connect the PC to both the buses.

9.7.1 Control Signals

Each register that writes to the write bus needs a control signal. If it is asserted (equal to 1), then the value of the register appears on the write bus. Otherwise, the value of the register does not get reflected on the write bus. For example, the register, $aluResult$, contains the result of an ALU operation, and it is sometimes necessary to transfer its value to the write bus. The signal $aluResult_{out}$ controls the behavior of the aluResult register. We associate similar signals with the subscript, $_{out}$ with all the registers that need to access the write bus.

Likewise, we associate a set of signals with the registers that are connected to the read bus. For example, the register mar is connected to the read bus. We associate the signal mar_{in} with it. If it is 1, then the value of the data on the read bus is transferred to mar. If $mar_{in} = 0$, the mar register is effectively disconnected from the read bus.

The PC has two signals associated with it: pc_{in} and pc_{out}. The μcontrol unit ensures that at one point of time only one register can write to the write bus. However, it is theoretically possible for multiple registers to read from the read bus concurrently.

9.7.2 Functional Unit Arguments

We augment the read bus to carry the arguments for the functional units (referred to as $\langle args \rangle$). These arguments specify the nature of the operation, which the functional unit needs to perform. For example, the two operations associated with the memory unit are $\langle load \rangle$, and $\langle store \rangle$, and the two operations associated with the register file are $\langle read \rangle$ and $\langle write \rangle$. Each ALU operation also has its separate code.

We propose to encode each operation in binary, and reserve the special value of 0 to indicate that no operation needs to be performed. Each functional unit needs to be connected to the read bus, and needs to process the value of the arguments. The $\langle args \rangle$ field can be split into two parts: $\langle$*unit id*$\rangle$, and $\langle$*opcode*$\rangle$. The $\langle$*unit id*$\rangle$ specifies the identifier for the functional unit. For example, we can assign 00 to the ALU, 01 to the register file, and 10 to the memory unit. The $\langle$*opcode*$\rangle$ contains the details of the operation to be performed. This is specific to the functional unit. We propose a 10-bit $\langle args \rangle$ bus that is a part of the read bus. We devote 3 bits to the $\langle$*unit id*$\rangle$, and 7 bits to the $\langle$*opcode*$\rangle$. Thus, for each unit we can support 128 different operations. Implementing the circuit to process the $\langle args \rangle$ is easy, and we leave it as an exercise to the reader.

9.8 The Microcontrol Unit

We now arrive at the last piece of our microprogrammed processor, which is the design of the μcontrol unit. It is a simple processor that executes microinstructions. It consists of a μfetch unit and a μpc. Every cycle we increment the μpc by 1 (addressed by the number of the instruction, not by bytes), or set it to the branch target. Then, we proceed to read the microinstruction from the microprogram memory, and process it. There are two main paradigms for designing an encoding of microinstructions, and executing them using a μcontrol unit. The first is known as *vertical microprogramming*. In principle, this paradigm is similar to executing regular program instructions using a hardwired processor. The second paradigm is known as *horizontal microprogramming*. This is more common, and is also a more efficient.

9.8.1 Vertical Microprogramming

In vertical microprogramming, we encode an instruction similar to encoding a regular RISC instruction in a hardwired processor.

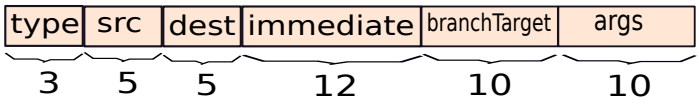

Figure 9.24: Encoding of a microinstruction (vertical microprogramming)

Figure 9.24 shows an encoding for our scheme. Here, we devote 3 bits for encoding the type of the microinstruction. We need 3 bits because we have a total of 8 microinstructions (see Table 9.8). Each microinstruction embeds the 5 bit id (because we have 19 registers visible to microprograms) of an internal source register, 5 bit id of an internal destination register, a 12-bit immediate, and a 10-bit branch target. At the end, we encode a 10-bit args value in the microinstruction. Each instruction thus requires 45 bits.

Now, to process a vertically encoded microinstruction, we need a dedicated μdecode unit that can generate all the control signals. These signals include all the enable signals for the internal registers, and the signals to select the right input in the transfer multiplexer. Additionally, it needs to extract some fields from the microinstruction such as the immediate, branch

target, and the *args* value, and subsequently extend their sign. We have already gone through a similar exercise for extracting the fields of an instruction, and generating control signals, when we discussed the operand fetch unit and control unit for our hardwired processors in Sections 9.2.3, and 9.3, respectively. The logic for generating the control signals, and extracting fields from the microinstruction is exactly the same. Hence, we leave the detailed design of these units as an exercise for the reader.

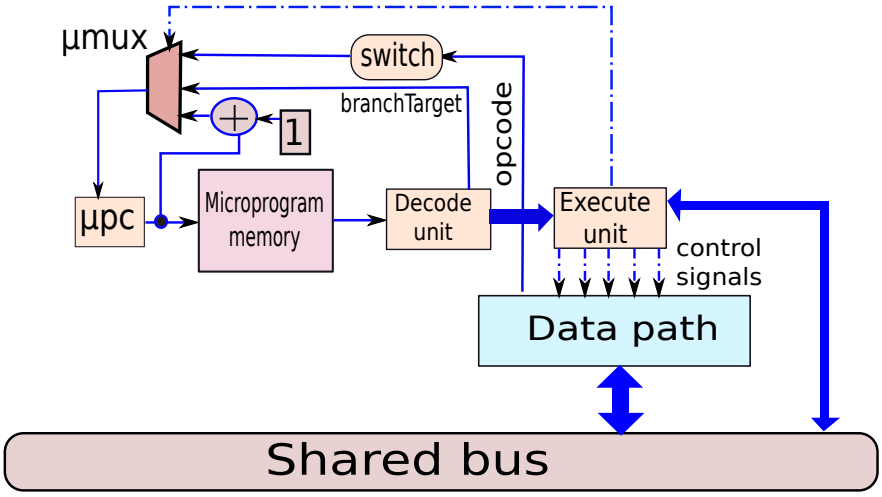

Figure 9.25: The μcontrol unit (vertical microprogramming)

The design of the vertical μcontrol unit is shown in Figure 9.25. We have a microPC (μpc), which saves the index of the currently executing microinstruction. Every cycle, we increment the μpc by 1. This is because each row of the microprogram memory saves 1 microinstruction. We assume that a row is wide enough to save the entire contents of the microinstructions. We do not have the requirement of saving data at the granularity of a fixed number of bytes here. After reading the microinstruction, we proceed to decode it. The process of decoding a microinstruction breaks it into a set of fields (instruction type, immediate, branch target, *args*, source, and destination registers). Subsequently, we generate all the control signals and dispatch the set of control signals to the execute unit. The execute unit sets all the control signals, and orchestrates an operation in the data path of the processor. The execute unit also sets the control signals of the transfer multiplexer. We need some additional support to process the *mswitch* instruction. We add a dedicated switch unit that takes inputs (the opcode) from the *ir* register, and computes the starting address for the microcode of the currently executing program instruction. It sends the address to the multiplexer, $\mu fetch$ (see Figure 9.25). The multiplexer chooses between three inputs – default microPC, branch target, and the address generated by the switch unit. It is controlled by the execute unit. The rules for choosing the input are shown in Table 9.9. In accordance with these rules, and the value of the $isMBranch$ signal (generated by comparing μimm, and the contents of the shared bus), the execute unit generates the control signals for the $\mu fetch$ multiplexer.

Instruction	Output of the $\mu fetch$ multiplexer
$mloadIR$	next μpc
$mdecode$	next μpc
$mswitch$	output of the switch unit
$mmov$	next μpc
$mmovi$	next μpc
$madd$	next μpc
mb	branch target
$mbeq$	branch target if $isMBranch = 1$, else next μpc

Table 9.9: Rules for controlling the $\mu fetch$ multiplexer

9.8.2 Horizontal Microprogramming

We can further simplify the design of the μcontrol unit. We do not need three steps (fetch, decode, execute) to execute a microinstruction. The decode step is not required. We can embed all the control signals in the microinstruction itself. It is thus not required to have a dedicated signal generator to generate all the control signals. By doing so, we will increase the size of the encoding of an instruction. Since the number of microinstructions is small, and we do not have any significant constraints on the size of the encoding of a microinstruction, adding additional bits in the encoding is not an issue. This paradigm is known as *horizontal microprogramming*. The encoding of a microinstruction is shown in Figure 9.26.

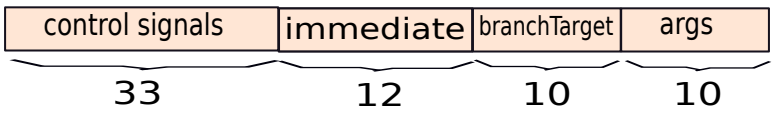

Figure 9.26: Encoding of a microinstruction (horizontal microprogramming)

We need the following fields – control signals (saved as a bit vector whose size is 33 bits), immediate (12 bits), branch target (10 bits), and args (10 bits). The reason we require 33 control signals is as follows. We have 19 registers (see Table 9.7) visible to microcode. Out of these register, the following 9 registers are exclusively connected to either the read bus or the write bus: ir, $flags.E$, $flags.GT$, I, rd, $rs1$, $rs2$, $branchTarget$, and $immx$. Hence, these registers require just one control signal. The rest of the registers have read-write functionality. Hence, these registers require two control signals. Thus, the total number of register enabling control signals are 29. We need 2 more signals each to control the transfer multiplexer, and the $\mu fetch$ multiplexer. We thus have a total of 33 control signals, and we require 65 bits to encode the instruction. Recall that with vertical microprogramming, we needed 45 bits.

Now, with additional storage we can completely eliminate the signal generator in the decode stage, and thus significantly simplify the μcontrol unit as shown in Figure 9.27

Here, we have eliminated the decode stage. All the signals are embedded in the instruction, and they are thus used to orchestrate a computation in the data path. The execute unit

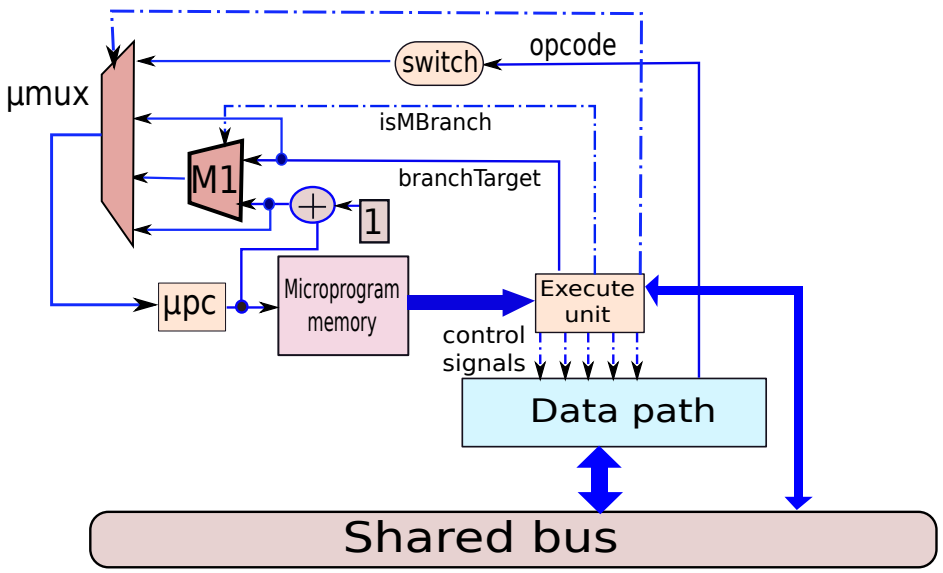

Figure 9.27: The μcontrol unit (horizontal microprogramming)

generates the $isMBranch$ signal (by comparing the μimm and the value on the read bus), which is used to choose between the next μpc, and the branch target using multiplexer, $M1$. Here, we slightly complicate the $\mu fetch$ multiplexer, and add a little bit of redundancy in the interest of simplicity. We make it a 4 input structure, and choose between the value from the switch unit, the branch target, the output of $M1$, and the next μpc. The 2-bit control signals for controlling the $\mu fetch$ multiplexer are embedded in the instruction in accordance with the rules given in Table 9.9. The rest of the operation of the circuit is the same as the circuit for vertical microprogramming as shown in Figure 9.25.

9.8.3 Trade-offs between Horizontal and Vertical Microprogramming

The trade-offs between horizontal and vertical microprogramming are the following:

1. Horizontal microprogramming requires more storage. However, this is not an issue in a microprogrammed processor. The additional storage is minimal.

2. Horizontal microprogramming eliminates the need for dedicated signal generation logic in the μcontrol unit.

3. To program a horizontally microprogrammed processor, it is necessary to expose the control signals to the programmer and the microassembler. This makes the microassembler very specific to a given processor. However, in vertical microprogramming, as long as the internal register set remains the same, we do not need different microassemblers.

To summarize, microprogramming is a very potent method to implement an instruction set. We can design very expressive instruction sets using this method. However, this is not

a preferable approach for implementing all the instructions (especially the common ones) in a high performance processor.

9.9 Summary and Further Reading

9.9.1 Summary

Summary 9

1. *We design a processor by dividing it into multiple stages, where the stages are mostly independent of each other. We divide our basic SimpleRisc processor into five stages: instruction fetch(IF), operand fetch(OF), execute (EX), memory access (MA), and register writeback (RW).*
2. *The roles of these stages are as follows:*
   (a) *The IF stage computes the next PC, and fetches the contents of the instruction, whose address is stored in the PC.*
   (b) *In the OF stage, we decode the instruction, and read its operands from the register file. Specifically, we compute the branch target, and expand the embedded immediate in the instruction according to the modifiers.*
   (c) *In the EX stage, we compute the branch outcome, branch target, and perform the ALU operations.*
   (d) *In the MA stage, we perform loads and stores.*
   (e) *Lastly, in the RW stage, we write back the values computed by ALU or load instructions, and the return address for a call instruction to the register file.*
3. *The data path consists of all the elements for storing, retrieving, and processing information such as the registers, memory elements, and the ALU. In contrast, the control path generates all the signals for controlling the movement of instructions and data.*
4. *We can use a hardwired control unit that generates all the signals for the control path.*
5. *For additional flexibility, and portability, we presented the design of a microprogrammed processor. This processor replaces every program instruction by a sequence of microinstructions.*
6. *We defined 8 microinstructions, and created a microprogrammed data path that connected all the units on a shared bus. Each unit in a microprogrammed data path exposes its input and output ports through registers. We use 19 registers in our design.*
7. *We subsequently showed implementations in microcode for all the instructions in the SimpleRisc ISA.*

8. *We designed a shared bus for such processors by interconnecting two physical buses (write bus, and read bus) with a multiplexer. The multiplexer (known as the transfer multiplexer) chooses between the output of the write bus, the output of the μadder, and the micro immediate.*

9. *We showed the design of a μcontrol unit for both vertical and horizontal microprogramming. Vertical microprogramming requires a decode stage for generating all the control signals. In comparison, horizontal microprogramming requires all the control signals to be embedded in the microinstruction.*

9.9.2 Further Reading

Processor design is very heavily studied in courses on computer architecture. Readers should first start with Chapter 10 that discusses pipelining. Chapter 10 is a sequel to the current chapter. The reader can then take a look at the "Further Reading" section (Section 10.12.2) in Chapter 10. In general, for basic processor design, the reader can also consult other books on computer architecture [Mano, 2007, Stallings, 2010, Henessey and Patterson, 2010] to get a different perspective. The books by Morris Mano [Mano, 2007], and Carl Hamacher [Hamacher et al., 2001] consider different flavors of microprogramming, and define their own semantics. If the reader is interested in the history of microprogramming per se, then she can consult books dedicated to the design and history of microprogramming [Carter, 1995, Husson, 1970]. The PTLSim [Yourst, 2007] simulator translates x86 instructions into micro-instructions, and simulates these microinstructions on a data path similar to that of commercial processors. Readers can take a look at the code of this simulator, and appreciate the nuances of processing and executing microinstructions.

Exercises

Hardwired Processor Design

Ex. 1 — We have divided a *SimpleRisc* processor into 5 distinct units. List them, and describe their functions.

Ex. 2 — Explain the terms – *data path* and *control path*?

Ex. 3 — How does having a lesser number of instruction formats help in the process of decoding an instruction?

Ex. 4 — Draw the circuit for calculating the value of the 32-bit immediate, from the first 18 bits of the instruction. Take the modifiers into account.

Ex. 5 — Why is it necessary for the register file in our *SimpleRisc* processor to have 2 read ports, and 1 write port?

Ex. 6 — Why do we need 2 multiplexers in the OF stage of the processor? What are their functions?

Ex. 7 — Let us propose to compute the branch outcome and target in the OF stage. Describe the design of the OF stage with this functionality.

* **Ex. 8** — For the ALU we use a multiplexer with numerous inputs. How can we implement this multiplexer with transmission gates? (show a circuit diagram, and explain why your idea will work)

Ex. 9 — Draw a circuit for implementing the *cmp* instruction. It should show the circuit for subtraction, and the logic for updating the flags.

Ex. 10 — How do we implement the *call* instruction in our processor?

Ex. 11 — Draw the circuit diagram for computing the $isWb$ signal.

Ex. 12 — Why do we use the $isAdd$ control signal for the load, and store instructions also?

Microprogramming

Ex. 13 — Compare a hardwired control unit and a microprogrammed control unit.

Ex. 14 — Draw the block diagram of a microprogrammed processor.

Ex. 15 — Why do we need the $mswitch$ instruction?

Ex. 16 — Describe the microcode implementation of the load and store instructions.

Ex. 17 — Write a program in microassembly to check if a number in register $r2$ is a perfect square. Save the Boolean result in register, r0.

Ex. 18 — Write a program in microassembly to check if the value in register $r2$ is a palindrome. A palindrome reads the same from both sides. For example, the 8-bit number, 11011011 is a palindrome. Save the Boolean result in register, r0.

* **Ex. 19** — Write a program in microassembly to check if the value in register $r2$ can be expressed as a sum of two cubes in two different ways. For example, 1729, is one such number. $1729 = 12^3 + 1^3 = 10^3 + 9^3$. Save the Boolean result in register, r0.

Ex. 20 — Outline the design of the shared bus, and microprogrammed data path. Explain the functionalities of each of its components.

Ex. 21 — Draw a detailed diagram of the μcontrol unit along with the transfer multiplexer in a vertically microprogrammed processor.

Ex. 22 — Draw a detailed diagram of the μcontrol unit along with the transfer multiplexer in a horizontally microprogrammed processor.

Ex. 23 — Compare the trade-offs between horizontal and vertical microprogramming.

Design Problems

Ex. 24 — Implement the hardwired *SimpleRisc* processor using Logisim, which is an educational tool for designing and simulating digital circuits. It is freely available at `http://ozark.hendrix.edu/~burch/logisim`. Try to support all the instructions, and the modifiers.

Ex. 25 — Now, try to implement a horizontally microprogrammed processor using Logisim. This project has two parts.

a) Write a microassembler that can translate microassembly instructions to their machine encodings. Use this microassembler to generate the microcode for all the instructions in the *SimpleRisc* ISA.

b) Create a data path and control path in Logisim for a horizontally microprogrammed processor. This processor should be able to directly execute the code generated by the microassembler.

c) Run regular *SimpleRisc* programs on this processor.

d) Implement custom *SimpleRisc* instructions such as *multiply-add* ($a \leftarrow b * c + d$), or instructions to find the square of a number on this processor.

Ex. 26 — Implement the basic hardwired processor in a high-level description language such as VHDL. You can use the freely available open source tool GNU HDL (`http://gna.org/projects/ghdl/`) to implement and simulate your circuit.