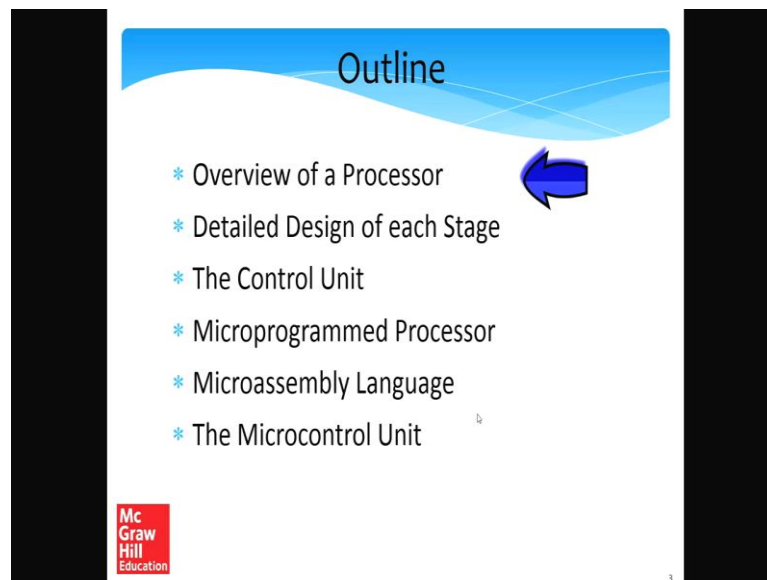


Computer Architecture
Prof. Smruti Ranjan Sarangi
Department of Computer Science and Engineering
Indian Institute Technology, Delhi

Lecture - 23
Processor Design Part-I

Welcome to chapter 8. It is finally, we are going to enter the gates of real computer architectures, where real computer architecture means that we will actually look at designing a processor. So, this is part of the chapter 8 of the book computer organisation and architecture published by Mc Graw Hill 2015. The important feature of this chapter is that we will look at designing a full working processor for the SimpleRisc instruction set from scratch. So, you will find models of this processor in both Logisim as well as in VHDL in the website of the book which is given over here. But the main aim is starting from first principles have been issue we want to design a full working processor that can run fairly complicated programs.

(Refer Slide Time: 01:58)

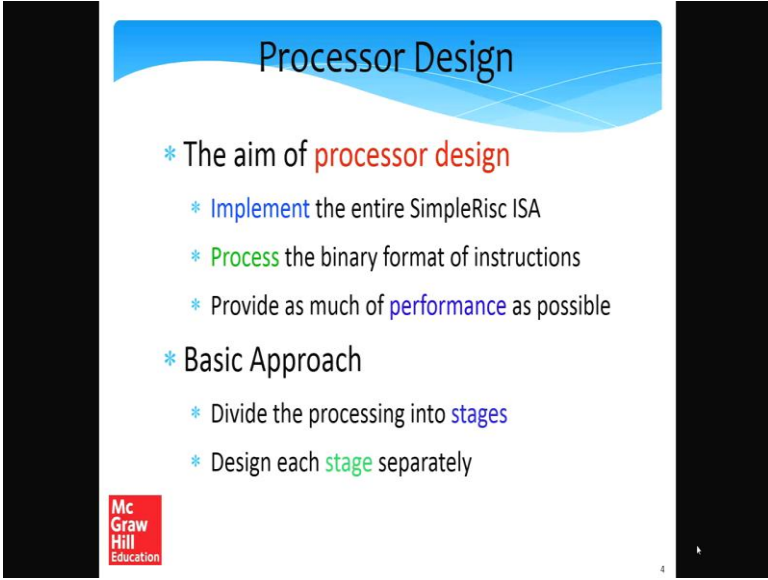


Let us now consider the outline of this particular chapter. So, we shall start with the overview of a processor, where we should describe all the features that are required to implement a single SimpleRisc processor. Then we shall look at the detailed design of each stage, focus on the control unit. After that and subsequently we shall describe a different type of processor that is you know it is the same processor designed for the same

ISA, but the style of designing it is very different. So, it is called a microprogrammed processor.

So, a lot of the current processors do have microprogrammed features. These are a very important thing to learn. So, a microprogrammed processor will have its own micro assembly language and its own micro control unit. So, we shall take a look at this towards the end of this particular chapter.

(Refer Slide Time: 02:53)



The slide is titled "Processor Design" in a blue header. It contains two main sections of bullet points. The first section, "The aim of processor design", lists three points: "Implement the entire SimpleRisc ISA", "Process the binary format of instructions", and "Provide as much of performance as possible". The second section, "Basic Approach", lists two points: "Divide the processing into stages" and "Design each stage separately". The slide also features the McGraw Hill Education logo in the bottom left corner and a small number "4" in the bottom right corner.


- * The aim of **processor design**
 - * **Implement** the entire SimpleRisc ISA
 - * **Process** the binary format of instructions
 - * Provide as much of **performance** as possible
- * **Basic Approach**
 - * Divide the processing into **stages**
 - * Design each **stage** separately

So, let us take a look at a processor design. So, the aim of designing a processor with transistors and all the logic gates that we have learnt and also using all the digital elements that we have learnt in chapter 6. So, the main aim is to implement the entire SimpleRisc ISA. So, we will get the instructions in the binary format in the 32 bit binary format. And we need to provide as much of performance as possible where performance is defined as you know inverse of the execution time of a program. So, we want programs to pretty much run as fast as possible.

So, the basic approach is to divide the processing of an instruction into several stages and look at each stage and design each stage separately.

(Refer Slide Time: 03:51)

A Car Assembly Line



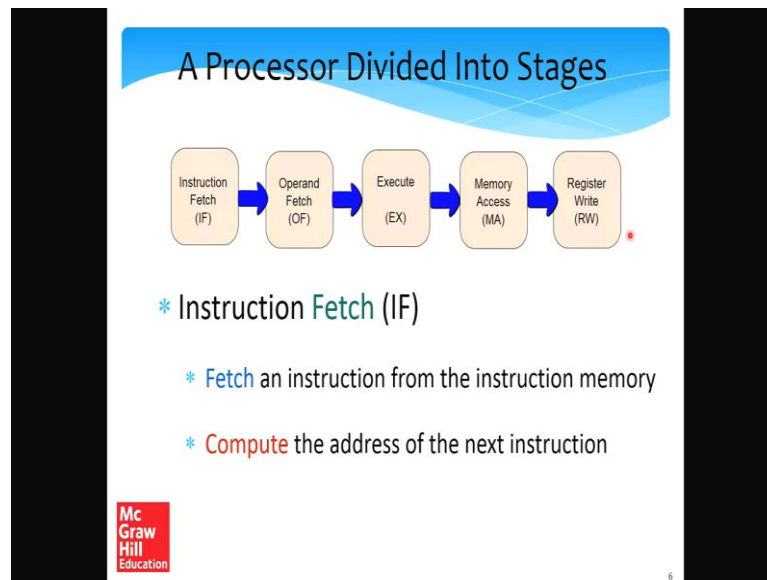
- * Similar to a car **assembly line**
 - * Cast raw **metal** into the **chassis** of a car
 - * Build the **Engine**
 - * Assemble the **engine** and the **chassis**
 - * Place the **dashboard**, and **upholstery**

Mc
Graw
Hill
Education

5

So, let us look at a car assembly line. So, in a car assembly line we typically have you know different shops right. So, we have one shop that cast raw metal into the chassis of a car chassis of a car. Then another shop builds the engines another shop builds the body another shop you know does dashboard and upholstery and finally, in a one group of people put everything together, and then at the end of a car's assembly line it goes through it is, finally painted right; painted and tested and then is ready for delivery. So, similarly an instruction also goes through various stages, at each stage some amount of processing is done to an instruction. And then from that stage it moves to the next stage subsequent stage it moves.

(Refer Slide Time: 04:49)



So, similarly we can divide a processor into multiple stages. So, we can. So, it is typical to divide the processing of an instruction into 5 stages. So, here we start with an instruction fetch stage. In the instruction fetch stage we fetch the instruction from memory. So, we will call this IF stage the instruction fetch stage is an IF stage. The next stage will be the operand fetch stage, that we shall call the OF stage where we fetch the operands be registers you know the registers not memory addresses, but registers and also in the blow up constants and immediate thing and all of that.

So, we will fetch the operand in this stage. In the EX stage EX standing for execute. We shall execute the instruction right run it. The 4th stage will be memory access stage where we will access memory perform a load or store. And finally, the fifth stage is will be called a register right stage where the results of the instruction will be written back to the register file. So, these are the 5 stages IF for instruction fetch, OF for operand fetch, EX for execute MA for memory access and RW for register write, it also known as the write back stage the last stage.

(Refer Slide Time: 06:44)

The slide features a blue header with the title "Operand Fetch (OF) Stage". Below the header is a list of five tasks, each preceded by an asterisk. The tasks are: "Decode the instruction (break it into fields)", "Fetch the register operands from the register file", "Compute the branch target (PC + offset)", "Compute the immediate (16 bits + 2 modifiers)", and "Generate control signals (we will see later)". The text "16 bits + 2 modifiers" is annotated with red handwritten notes: "16 bit" above "16 bits", "32 bit" below "16 bits", and a circled "16" to the right. A red arrow points from the circled "16" to the "32 bit" text. In the bottom left corner, there is a red logo for "Mc Graw Hill Education".

- * **Decode** the instruction (break it into fields)
- * **Fetch** the register operands from the register file
- * **Compute** the **branch target** (PC + offset)
- * **Compute** the **immediate** (16 bits + 2 modifiers)
- * **Generate** **control signals** (we will see later)

Mc Graw Hill Education

So, the instructions fetch stage IF stage what does it do, it fetches an instruction from instruction memory. And it computes the address of the next instruction. The operand fetch OF stage does several things. First is, decode the instruction. Decoding the instructions means you break it into fields where everything is packed inside 32 bit field. So, we need to break it into fields. Then we need to fetch the register operands from the register file. So, we need read them. So, this stage also computes the branch target which is adds the offset to the program counter.

So, if any of these terms short of looks for in then the leader is requested to go back to chapter 3 and take a look at a SimpleRisc ISA once again. Because we are pretty much designing processor for the SimpleRisc instruction set. So, it is important for the reader you know if the reader forgotten to go back to chapter 3 and sort of recapitulate these terms. So, what we shall do here in the several things first is decode fetch the operands to the register file, compute the branch target which is essentially the next the id of the instruction after the branch.

So, in this case we add the offset to the program counter. We also in this stage compute the immediate. So, the immediate mind you in SimpleRisc is 16 bits plus 2 modifiers. So, it is an 18 bit immediate in that sense. So, everything inside the processor is 32 bits pretty much this 18 bits need to be in a sense expanded and grown up to a 32 bit quantity. So, this is called expansion of the immediate.

So, we need to compute the immediate, which is 16 bits plus you know 2 bit modifier. So, I shall call it 2 modifiers I should call it 2 bit modifiers that is slightly get a slightly better. So, we also have 2 modifiers. So, that is also not wrong. So, 2 modifiers are we have a u modifier, but we treat the 16 bit quantity as unsigned, and we have the h modifier where we essentially the 32 bit constant that we create in that the 16 bits are loaded in the upper 2 bits. So, this immediate that we have, that needs to be expanded after taking the modifiers into account into a 32 bit quantity. So, the default where we as to extend sign, but we can over write this default behaviour with the u and h modifiers in the instruction. And finally, we also need to generate controls signals.

But we shall see that later the main aim of the OF stage is dose understand the instructions fetch the operands compute the 32 bit branch target and immediate after taking the immediate modifiers into account.

(Refer Slide Time: 09:59)

The slide is titled "Execute (EX) Stage" in a blue header. Below the title, there are three bullet points:

- * The EX Stage
 - * Contains an Arithmetic-Logical Unit (ALU)
 - * This unit can perform all arithmetic operations (add, sub, mul, div, cmp, ^{mod}or), and logical operations (and, or, not)
 - * Contains the branch unit for computing the branch condition (beg, bgt) >
 - * Contains the flags register (updated by the cmp instruction)

 The slide also features the McGraw Hill Education logo in the bottom left corner and a small number '8' in the bottom right corner.

Now let us take a look EX stage. So, EX or the execute stage contains an arithmetic logical unit and ALU. So, the unit can perform all arithmetic operations add subtract multiply divide compare or and so basically all well this you actually be a instead of or it should actually be a mode. So, it can find essentially the do an division and find the remainder. And all the logical operations which are and or and not. So, the execute stage also contains the branch unit for computing the branch condition.

Basically, if you would recall; we have 2 branch conditional branch instructions in the SimpleRisc called beq in bgt. So, beq basically means branch if equal and bgt means branch if greater than. So basically if one of the last comparisons not one of the last if the last comparison has resulted in equality then the beq condition will true. Otherwise the if the last comparison as resulted in a greater than then the bgt condition will be true. So, this is also done in a EX stage, and EX stage also contains the flags register which is updated by the compare instructions.

So, just to summarize the execute stage contains the arithmetic logical unit primarily. So, the ALU does arithmetic operations add subtract multiple divide compare it finds the remainder the mode. And it does logical operations which is and or and not. Along with that it processes the branch conditions for the beq and bgt instructions branch be equal and branch if greater than instructions.

So, in this case it essentially reach the flag register to find out the result of the last comparison and on the basis of that decision is taken.

(Refer Slide Time: 12:17)

MA and RW Stages

- * MA (Memory Access) Stage
 - * Interfaces with the memory system
 - * Executes a load or a store
- * RW (Register Write) Stage
 - * Writes to the register file ✓
 - * In the case of a call instruction, it writes to register, ra

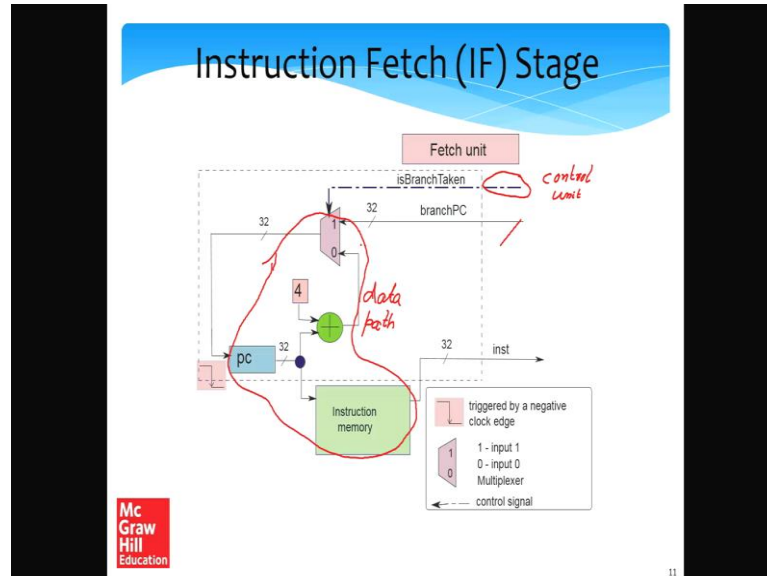
McGraw Hill Education

call
ra

Subsequently we have the MA memory access stage which interfaces with the memory system and executes the load or a store. And we also have RW stage at end the fifth stage, which writes to the register file and in the case of a call instruction when we are calling. So, recall that in the cases of a call instruction address of the next instruction was saved in the written address register ra. So, it will basically write it is try to registered ra

what does it write the address of the next instruction here their address of the written address is written into register ra.

(Refer Slide Time: 13:11)



So, now let us consider the detail design of each stage. So, let us take a look at circuit diagram of IF stage of the fetch stage. So, we will also see its fetch unit. So, in this case we will be showing the circuit diagram will be seeing similar circuit diagrams and almost for almost all the stages say important to have an understanding other way that is diagrams are drawn.

So, essentially each box determines is either a storage unit or a computer unit. And we will be using this sign for a multiplexer. So, recall a multiplexer is a logical unit that chooses one of the 2 inputs. So, at a negative edge of clock, we assume that all are circuits or negative edge triggered. So, whenever that clock goes from positive to negative a computation starts. So, the pc we will call the pc as a program counter register it is 32 bit register. So, the pc flows to the instruction memory the instruction memory can be an, it is typically any array of sram cells. So, we read. So, we will be discussing on how this instruction memory is created in chapter 10, but for the time being let us assume that you know it is a black block that we do not have touch and which is per say out of fetch unit.

So, we read an instruction from it which is again a 32 bit quantity. So, we will be using this symbol over here to indicate that the wire is carrying 32 bits. So, we will read the instruction and simultaneously we will consider the program counters address and add 4

to it. So, why will we add 4 to it, because we want to get the address of the next instruction. Of course, same means not a branch and so since each instruction is 4 bytes and we address memory at the level of the bytes will add 4 to it. So, this will be the one of the inputs to the multiplexer.

The other input to the multiplexer will be the program counter pc or the branch called branch pc, which is the 32 bit quantity. So, if the instruction is a branch the target needs to be chosen right which we are calling the branch pc. Or if it is not a branch we need to choose the next instruction which is the current pc address plus 4. So, the multiplexer will choose one of the addresses depending upon whether the instruction is a taken branch or not and there will be. So, we will be using the dotted lines here to indicate the control signals where controlling the multiplexers.

So, we are assuming a signal called each branch taken which is coming from another stage. This will say if whether if it a taken branch then essentially this value will flow out. If it is not taken branch a not a branch, then pc plus 4 is what flows out. So, these controls the multiplexer on in one of the input will show up at the output. And this again will be the value of the next program counter that will be used in the next cycle for the next instruction. So, this is the fetch unit the fetch unit per says fairly simple. The important points to note are that we will be using this dotted arrow for the control signals for controlling the multiplexers. And we will be using the multiplexer we have a 0 input and one input basically it is say that if each branch taken one we use the branches pc.

Otherwise we use pc plus 4 as the input which is show up at output. So pretty much that is all it is a fairly simple circuit, with one register one add and one multiplexer. So, that almost you know as simple as it gets.

(Refer Slide Time: 17:14)

The Fetch unit

- * The **pc** register contains the **program counter** (negative edge triggered)
- * We use the **pc** to access the instruction memory
- * The **multiplexer** chooses between
 - * pc + 4
 - * branchTarget
- * It uses a control signal → isBranchTaken

Mc
Graw
Hill
Education

12

So, the fetch unit I just want to repeat contains a register called the pc register, which is negative edge triggered. And the multiplexer chooses between pc plus 4 the contents of pc plus 4 and the branch target, which is called the branch pc and it uses a control signal called isBranchTaken.

(Refer Slide Time: 17:37)

isBranchTaken

- * isBranchTaken is a **control** signal
 - * It is generated by the EX unit
- * **Conditions on isBranchTaken**

Instruction	Value of isBranchTaken
non-branch instruction	0
<i>call</i>	1
<i>ret</i>	1
<i>b</i>	1
<i>beq</i>	branch taken - 1
	branch not taken - 0
<i>bgt</i>	branch taken - 1
	branch not taken - 0

Mc
Graw
Hill
Education

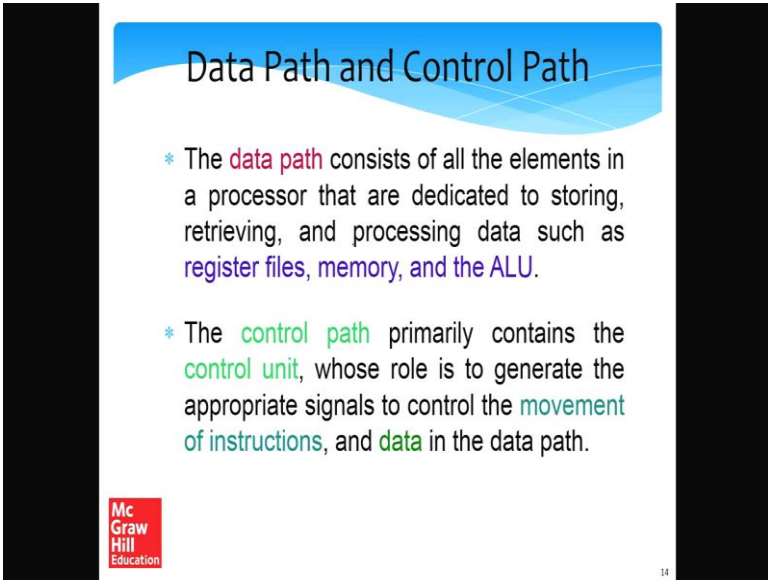
13

Let us now consider the isBranchTaken signal. So, the isBranchTaken as a control signal which is generated by EX units. So, its job is to control the multiplexer in the IF stage. So, basically this is how isBranchTaken is set. So, if it is a non-branch instruction. So, the

value of the isBranchTaken is 0 it is not taken branch. For these 3 kinds of instructions call ret and b and unconditional branch these are always taken. So, isBranchTaken is 1, for beq and bgt instructions which are conditional branch instructions if the branch is taken then then isBranchTaken is equal to 1 as we can see over here and if the branch is not taken then the value of the isBranchTaken signal is 0.

So, isBranchTaken basically says whether it is a taken branch are not if it is not taken because beq and bgt evaluate to falls, or it is not a branch instruction the value of the isBranchTaken is 0 otherwise it is 1.

(Refer Slide Time: 18:55)



The slide features a blue header with the title "Data Path and Control Path". Below the title, there are two bullet points. The first bullet point, marked with a red asterisk, defines the data path. The second bullet point, marked with a green asterisk, defines the control path. At the bottom left of the slide is the McGraw Hill Education logo, and at the bottom right is the number 14.

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

McGraw Hill Education

14

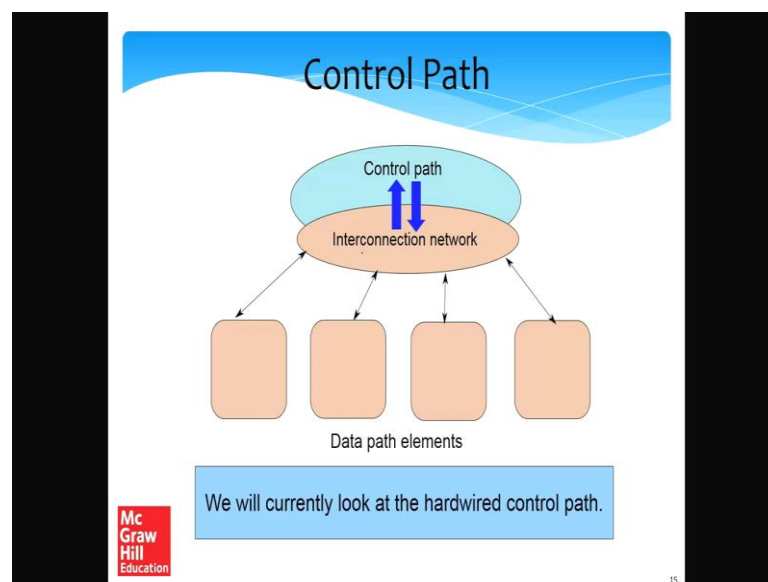
So, let us now define 2 very important concepts that we shall keep seeing over and over again in different form, where the data path and the control path. So, the data path essentially consists of all the elements in processors that are dedicated to storing retrieving and processing data such as a register files for keeping the register the memory the ALU and also in this case program count the register and so on.

So, the control path is essentially what consists of the dotted lines. So, the control path primarily contains the control unit whose role is to generate the appropriate signals to control the movement of instructions and data in the data paths. So, let us come back to this diagram. So, in this diagram all of this is there in the data path right all of this. What is there is the control. So, this is like a data path right. All of this which essentially consists of the signal to control this multiplexer, and also the control unit that generates

this signal they consist of they comprise the control path. So, the control unit and this by these wires you control the multiplexer their path of the control path. So, whether the multiplexer per say should be a path of the data path or the control path. Well that depends on interpretation, but we would like to keep it in the data path, because that is like the data path is like in a static aspect of this circuit.

So, consider the data path is the network of the roads, and consider the control path as a traffic light control or the traffic lights. So, network road pretty much static entity whether traffic lights control how the cars flow in the networks. So, we can think of the control path is something similar inside a processor.

(Refer Slide Time: 21:20)



So now, that we have the data path and the control path; so the control path as a control unit; and it as an interconnection network where it senses the wire to all the elements inside the data path.

So, this contains the controls movement of data and movements of instructions values in an everything inside the data path. Similarly, to what we have in a traffic network, where is traffic network is a data path the cars as instruction themselves and the controllers for the traffic lights and so on are part of the control path. So, currently we are looking at what is called hardwired control path. In the sense that the control path is fixed it cannot be changed it is made in silicon and it cannot be changed you know. So, that staticness to the control path is there that it cannot be modified, but when we look at a

microprogrammed processor, we will see that the aspects of the control path will be exported to the programmers.

So, in a sense the control path can be modified and controlled. So, we take a look at later, but let us take a look at this approach which is a by far more common which is to have a hardwired control path which means that one the chip as been designed and fabricated the control path cannot be changed. So, it cannot be reprogrammed.

(Refer Slide Time: 22:43)

Operand Fetch Unit

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

OpCode (handwritten in red above the table)

18 (16 bit, 2 bit) (handwritten in red above the table, with an arrow pointing to the 'Format' column)

Mc Graw Hill Education logo is present at the bottom left of the slide.

So, let us take a look at the operand fetch unit now. So, we have the finish instructions fetch unit it is the second stage the operand fetch unit. So, let us just take a quick look at SimpleRisc instructions. So, these are all our arithmetic instructions that we have add subtract multiply divide and get the modulo of the remainder, all of them have the same format we first have the destinations registered.

Then we have the first source register, subsequently we have the second source which can either be a register or an immediate value. The immediate value is actual an 18 bit quantity, 18 bits are divided into 2 fields 16 and 2. So, it is a 16 bit constant and we have space for 2 modifiers plus the default one; so 3 modifiers. So, that takes 2 bits. Then we have the compare instruction. So, there actually should be a space here right. So, it is compare space rs 1 comma. So this is rs 1 register source 1 and rs 2 slash immediate is either a register source or an immediate constraint value.

Then we have some logical instructions and on or and not which have exactly the same format as the arithmetic instructions. We have the move instructions to move data of from a register or an immediate into another register. These 3 lsl lsr asr are the 3 shift instructions they have exactly the same format rest of the arithmetic instructions. We have a no op instruction which does absolutely nothing. The benefit of a no op instruction will be cleared in chapter 9, which is a next chapter on pipelining. Till that point will not be able to understand; what is a greatness of a no op instruction.

The load and store instructions over here load is use to read something from memory and the store instruction is use to write something back to memory. So, they have the same format. So, mind you we made an exemption for store. In the sense of store actually does not have a register destination this destination is memory, but we did not want to change the format. So, we use the same format we use rd as the register to which we want to load and we use rs as a source register.

In the case of store we shall see that this will slightly complicate our processor design not a lot, but slightly we will use the immediate as the constraint offset and use rs 1 as the base register. We have 5 branch instructions. So, beq bgt and b are the 2 unconditional 2 sorry 2 conditionals and one unconditional branch instruction. So, beq and bgt require we need to know the result of the last comparison and b is an unconditional branch. Call is for function call and ret is for returning from the function.

So, let me now add a word of y. So, for every instruction we assign it a code. So, how many instructions we have we have 1 2 3 4 5 6 7 8 9 10 11 this is empty we have 21 instructions. So, we have 21 codes for the different instructions and so since we have 21 instructions we require a 5 bits. So, that a reason we have 5 bit code in this code is also called opcode there is a traditional name for the code of an instruction opcode or operation code. So, this 5 bit opcode that we have uniquely identifies an instruction. So, the question is why have we given the opcodes to the instructions the way that we have.

So, let us take look let us take a look at branch instructions first. So, essentially we need to look at decode complexity. So, we take a look at a branch instruction, we will see that all the branch instructions that we have has been given an MSB of 1. So, in that sense it is very easy to find different if an instruction is a branch or not. We just need to take a look at the most significant bit of the opcode and see if it is 0 or one. So, that is very easy

to find out if an instruction is a load or a store you will see that they been given consecutive opcodes.

So, we actually need to verify and look at the first 4 bits. If the first 4 bits as 0 1 1 1 then we will know it is a load or a store. So, that will also tell us that we need to access the memory access stage. And similarly we have a quick way of finding if you know an instruction is a shift or something. And so that essentially you know you know in opcode design we have to look at how easy to it is to understand and how quickly we can decode an instruction understand it. So, it is always a good idea to group similar instructions give them similar opcodes.

For example, all the branch instructions we have given them in MSB most significant bit of 1, which is made or instruction design in a sense actually very easy, is very easy to find out if an instruction is a branch are not. We just take a look at it is MSB most significant bit, and for load and store also we have to take look at first 4 bits, but that is also easy that is not very hard to do.

(Refer Slide Time: 28:57)

The slide is titled "Instruction Formats" and contains a table with the following content:

Format	Definition
branch	op (28-32) offset (1-27)
register	op (28-32) <u>1(27)0</u> rd (23-26) rs1 (19-22) rs2 (15-18)
immediate	op (28-32) <u>1(27)</u> rd (23-26) rs1 (19-22) imm (1-18)

Below the table, there are handwritten annotations in red ink:

- A note: *op* → opcode, *offset* → branch offset, *I* → immediate bit, *rd* → destination register
- A note: *rs1* → source register 1, *rs2* → source register 2, *imm* → immediate operand
- A diagram for the register format: $st\ rd, imm[rs1]$ with arrows pointing from *rd* to *rd* and from *imm[rs1]* to *rs1*.
- A diagram for the immediate format: $st\ rs1, 20[rs2]$ with arrows pointing from *rs1* to *rd* and from *20[rs2]* to *rs1*.

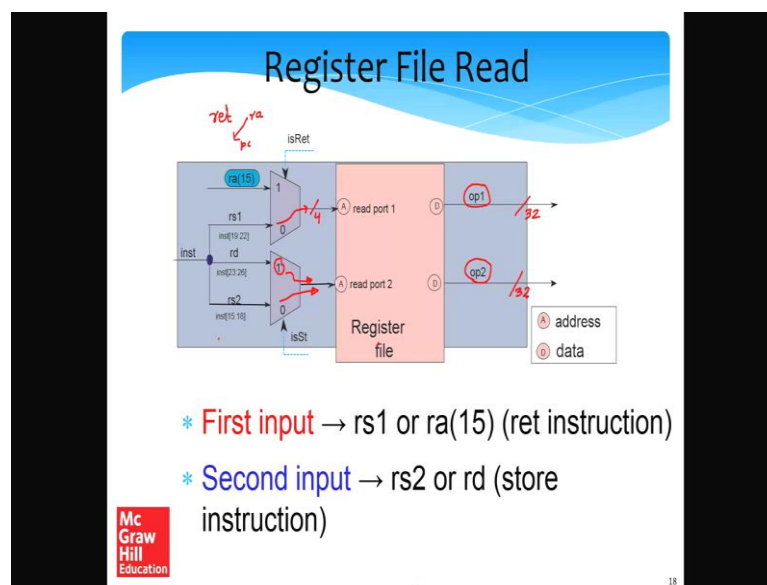
* Each format needs to be handled separately.

McGraw Hill Education logo is visible in the bottom left corner.

So, the instruction format set we have is we have 3 instructions formats; so the branch format. So, always the 5 bit opcode is the top 5 bits, bits 28 29 30 31 32 that is always the case.

We then have in the case of a branch we have a 27 bit offset. So, it is actually word offset not a byte offset for a word is units of 4 bytes. Then we have the register format. So, in the register format we specify whether it is an immediate and also the immediate bit is actually set to 0. So, I can sort of write 0 over here and I can write one over here for the immediate bit. Then we have the register destination register source 1 register source 2, in the case of an immediate format which is exactly the same at least the first part, we have the register destination the first source register and then we have the 18 bit immediate which is again divided into 2 fields.

(Refer Slide Time: 30:07)



So, each format needs to be handled by the decode circuit in the OF stage separately. So, now, let us take a look at. So then the next part after doing a little bit of decoding in the OF stage is to read the contents from the register file. So let us take a look at what is happening. So, given an instruction if we consider bits 19 to 22; so 19 to 22 is rs 1 and 15 to 18 is rs 2. So, what we do is we assume register file. A register file is basically a set of registers. So, that is called a register file. So, we have 16 registers in a SimpleRisc.

So, the register file will contain 16 registers. So, here it will have 2 read port read port basically means that 2 register ids can be given to the read port and we can sort of read the values of 2 registers simultaneously in a single cycle. So, a is for the address of the register which can be any number between 0 and 15 and d is for the data that was there in the register. So, the address is typically a 4 bit quantity between 0 and 15 and the data is

a 32 bit quantity which contains of the register. So, we will call the 2 contents op 1 and op 2. So, the idea is that given an instruction we will have 2.

So, let us consider the first input. So, in the first input we will either we will have either rs 1 which is the first source register or we need to take a look at. So, let us take a look at the ret instruction the written instruction. So, what the written instruction does is that it reads the value of the register ra or in a register r 15 right ra or r 15 same things. So, basically the contents of register ra are transfer to the pc, that is exactly what the written instruction does that whatever address is stored inside the written address register, that is becomes the contents of the program counter in the next cycle.

In affect the function returns and go backs to the instruction just after the call instruction. So, it is necessary to read the contents of ra. So, that a reason we have multiplexer, inside the multiplexer if this is ret signal is true is ret means is returned. So, if this signal is true with this means that this is the written instruction a ret instruction. So, we choose ra which is 15 as the output of the multiplexer if the instruction is not a written instruction we choose rs 1, which as the output of the multiplexer. So, once again what does this multiplexer do, it chooses between rs 1 which is the first source register and it chooses between 15 which is the default source for the ret instruction and mind you there will be no the ret instruction does not need rs 1. So, we can get way this.

So, this goes to the address port of the read port 1 of the register file and then we read the data out. And this value will refer to as op 1. Now we let us take a look at second multiplexer. So, mind you we had done special case. So, second multiplexer will be a default input which is rs 2, the second source register going back the see that the second source register is bit is between bits 15 to 18. So, that is the idea to the second source register, but also in the case of a of the store instruction we had actually done something very interesting in a immediate format we have assumed that for the case of the store register we will actually have the store register sorry the store instruction we will actually write it in this form store rd immediate rs 1. So, store did not have an rs 2 register source to field and rd is pretty much source it is not a destination.

So, we are reading the content register rd and saving at a memory right. So, the store instruction has actually 2 register sources, which are rd and rs 1 right. So, these are the 2 register sources of the store instruction. So, the complexity that this introduces is that we

need to add a multiplexer. If the instruction is a store instruction, we will then consider this output with 1, we will read an rd. So, mind you rd is this instruction output here which. So, basically let us give one more example still not clear. So, let us assume we have an instruction of the form store r1.

So, in this case r1 is the source register which needs to be read, and then we need to compute the memory address by taking a content of r2 and adding 20 and we need to affect the store. So, the field in which r1 which is 0001 will be stored this field is rd in instruction. And r2 will be stored in the field which is this field rs1. So, basically the sources in this stored instruction rd which is bits 23 to 26 and rs1 which is bits 19 to 22. So, that is the reason the store instruction will use the rd field which is bits 23 and 26 as the input for the register file read port 2. If it is not store instruction, then input will be the second source register which is rs2.

So, mind you it is always possible that we are only using we have only one register source. In this case we have doing some amount of extra work in producing junk value. So, that is still fine because this is not going a correct output of the instruction itself. So, doing a little bit of an extra work in the name of simplicity is still fine. So, once we have done this. So, second input as you can see either be rs2 or rd basically take store instruction into account we read the value and the value will be a 32 bit quantity which we shall hence forth refer to as op2. So, to summarize in this slide we are showing the register the circuit for reading the register file. So, we needed to add 2 multiplexers to the left of the register file to basically take some special cases into account.

For the first read port read port 1 we took into account the fact that the isret instructions reads a written address register. So, we need to choose between the written address register in the first register source rs1. So, that is the reason we have multiplexer is with the isret signal. Similarly, for read port 2 we have a multiplexer with the a isstore signal we choose between rd or rs2 if it is a store rd is a source which needs to ret and if it is any other ALU instruction then the rs2 is the source that needs to ret. So, then we read it from read ports 1 and 2 simultaneously and the data comes out which are refer to an op1 and op2 respectively.

(Refer Slide Time: 38:40)

Register File Access

- * The **register file** has two **read ports**
 - * 1st Input
 - * 2nd Input
- * The two outputs are **op1**, and **op2**
 - * **op1** is the **branch target** in the case of a **ret** instruction, or **rs1**
 - * **op2** is the value that needs to be stored in the case of a **store** instruction, or **rs2**

Mc
Graw
Hill
Education

19

Let me now repeat what was said in the previous slide. So, the register file has 2 read ports. So they take the inputs. The first read port whose output is op 1 is the branch target in the case of a ret instruction. So, in the written instruction that is actually the written address branch target or the written address in this case or it is the first source register rs 1, op 2 is the value that needs to be stored in the case of store or it is the second register source rs 2.

(Refer Slide Time: 39:24)

Immediate and Branch Unit

Mc
Graw
Hill
Education

20

- * Compute **immx** (extended immediate), **branchTarget**, irrespective of the **instruction format**.
- * For the **branchTarget** we need to choose between the **embedded target** and **op1** (ret)

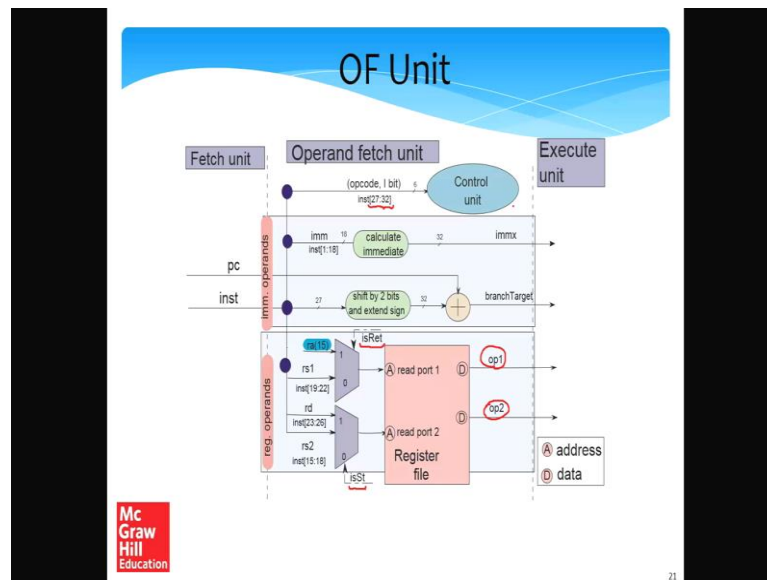
So, now let us take a look at immediate and branch unit whose job is to compute the branch target or the immediate. So, this is in the OF stage.

So, what we have coming in from the IF stage is the instruction. So, from the instructions what we do is that for the case of a branch we remove the opcode. So, 27 bits are left. Since this is the word offset not a byte offset we shift the number by 2 bits and extend the sign, so by shifting it by 2 bits. So, reason we do it is because is the offset in terms of memory words and one memory word is 4 bytes. So, what we do it we take 27 bit quantity and we shifted to the left by 2 positions. So, we get 29 bit quantity and then we extend it is sign to make it 32 bit number. So, this is essentially the offset in terms of bytes we add this to the program counter to give us the branch target. So, the branch target is the program counter plus the offset and mind you the offset was in terms of memory word. So, this was made offset in terms of bytes and it was extended to 32 bits.

So, in this case we compute the branch target. So, mind you we do this even for non-branch instructions because at this point we do not know for sure at instruction branch or not a branch. So, it always helps if we do some amount of extra work which might get wasted if the instruction is not a branch, but never the less in the interest of time it is needs to be done. So, it little bit of wasted work, but it is still fine the case of an immediately slightly more complicated. So, we again extract the bottom 18 bits. This is done for all instructions irrespective of whether it has an immediate or not because of this point do not really know if an instruction has an immediate or not.

So, what we do is we calculate the immediate. So, the way that we do is that we considered the 16 bit constant and the 2 bit modifiers based on that we create a 32 bit number. So, just to repeat if it is a default modifier then what we do is that we take the 16 bits specified and we essentially extend it is sign with the sign bits extended. If these are default modifiers, if it is the unsigned modifier what we do is that we write the 16 bits in the lower 2 bytes and in the upper 2 bytes we just write zeros, if it is the h modifier what we do is that we write the 16 bits in the upper 2 positions and in the lower 2 bytes we write zeros. So, in this way we create a 32 bit constant by extending the immediate and we call it immx. So, these fields are computed for all instructions might content junk in the case of some instructions. And after doing that we are done with a OF unit.

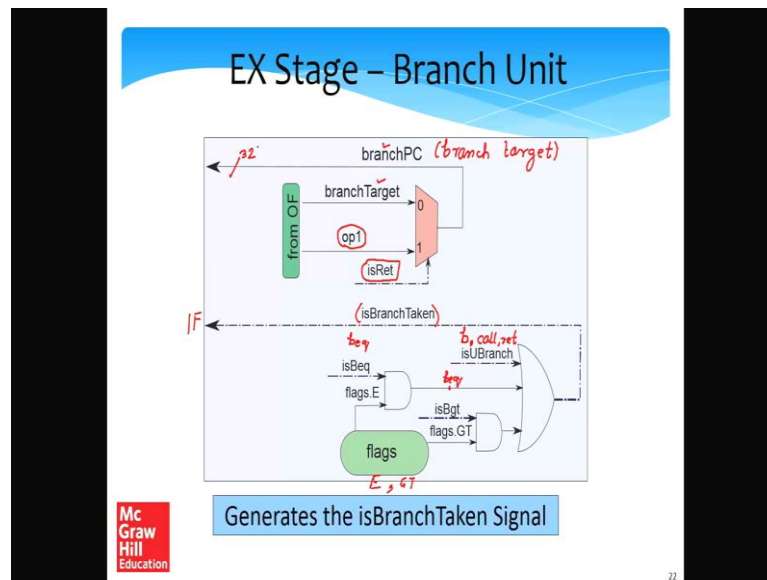
(Refer Slide Time: 42:44)



So, the way it stands is that we have the fetch unit the instruction fetch unit on one side. And we have the execute unit on the other side. So, and we have 2 main component or sub systems in the operand fetch unit. So, the first is the register file. So, register file contains the registers. So, in that we read the values of the register it takes 2 control signal as input isret and is store. And there are 2 outputs op 1 and op 2. So, depending upon the instruction they can contain different things. So, in the case of a normal ALU instruction they contain the first and second registers source operand. And if it is a written instruction op 1 will contain the written address, and if it is a store instruction op 2 will contain the register whose value needs to store and op 1 contain the base register.

So, in any case let us refer to these fields as op 1 and op 2, then we have a branch in immediate unit where we compute the branch target by adding the offset to the pc and we compute the extended immediate the immediate 32 bits by expanding the 18 bit that we embedded in the instruction. Additionally, what we do is that we need to compute all of these control signal isret store and isret and so on. So, we send the opcode and the immediate bit 6 bits to the control unit whose job is to generate the signals. So, the assumption is that control unit very quickly generates isret and isstore. So, I said they can be used in the same stage and for other stages it can take little bit more time.

(Refer Slide Time: 44:42)



Now, let us take a look at the execute stage the EX stage which also contains multiple units. So, let us first take a look at the branch unit which is very simple. So, let us start from the bottom. So, let us look at the flags register which content mainly 2 bits the e bit and the gt bit. So, the e bit is set if the last comparison led to an equality and the gt bit the greater than bit is set if last comparison led to a greater than condition that you know the first operand greater than second. So, now, let us see. So, if the instruction is a beq branch if equal instructions then the isbq signal is set 1 set to true. So, if this signal is set to true and flags dot e the last comparison had led to an equality.

Then this means that this should be a taken branch. So, we have an and gate and this specific one at value on the specific one indicates if it is a taken branch because beq is true. So, we can say the beq branch is equal this condition is true because it is a branch if equal instruction that is number 1 and the number 2 the last comparison led to an equality. Similarly, we can do the same for the bgt instruction, we can see if the last instruction if the current instruction is the bgt or not this signal set by the control unit and the value comes from the flag registers. So, we do and so the and output of the and gate tells us whether the instruction is bgt and if the last comparison resulted in greater than see both of them are true then only the output is true, and we consider one more control signal is u branch u branch means unconditional branch.

So, what will be an unconditional branch a branch instruction just a b instruction is a non-conditional branch if we have a call or written or something then also it is an unconditional branch right. So, in that case, but we look at call and written later let us only consider the b instruction for the time being is u branch. So when we discuss call and return bill again come back to this slide. So, basically if that is true which means you are branching in any case. So, we do an or of this. So, an or gate is needed to basically find out whether this is a taken branch or not if it is a taken branch right if you know we are branching to some other location. Then this control signal is set to true. So, mind you this control signals isBranchTaken which is going to the higher stage.

So, let me go back to the higher stage and show you where exactly it was being use. So, in the higher stage if you see this isBranchTaken signal which is essentially coming from the EX stage is being use to set the value of this multiplexer. So, either we choose pc plus 4 or we choose the branch pc. So, let us go back. So, in this case what we see is that the isBranchTaken will be set if dq or bgt conditions are true or if it is a unsigned branch and an unsigned branch can. So, what are the instructions which are sorry not an unsigned unconditional branch.

What are the instructions for which an unconditional branch condition is true is either b or a call or a ret right? So, that is when we know that we are changing the value of the pc for sure. So, basically that is when this condition will be true, and this conditional generated by the control unit. Now let us take a look at this multiplexer over here. So, let us assume an is ret signal which is again generated by the control unit, which as if an instruction is a written or not. So, if an instruction is of a type written then essentially we use op 1 as the branch pc.

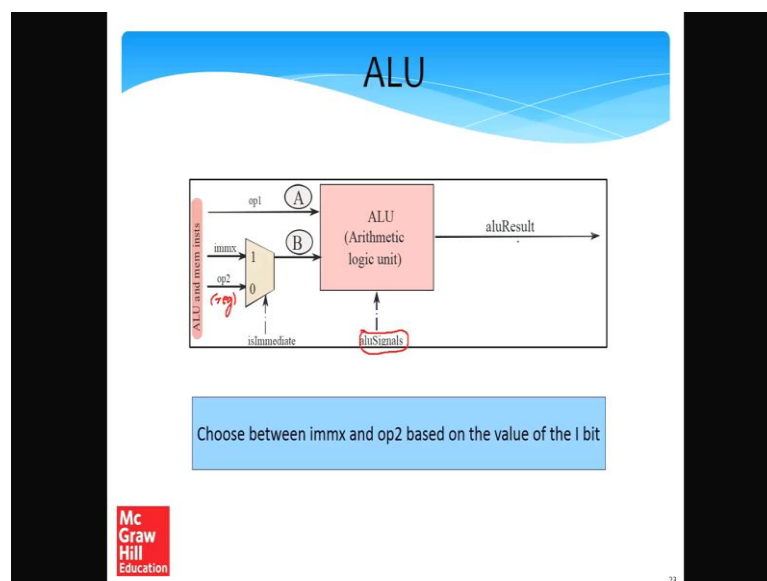
So, branch pc can be thought of as the branch target, write this is the same thing the target of the branch where you are branching 2. So, in this case if you see op 1 if it is a written instruction then essentially op 1 is the contents of the written address register which is the essentially the written address. So, if it is a written address instruction we use a written address as the target otherwise, we use the branch target which essentially comes from the previous stage the OF stage we use the branch target which the pc plus the offset as the branch pc or as the branch target. So basically we are differentiating between the terms branch target and branch pc. The reason we are doing that is because branch target is something that we are computing by adding the offset to the pc. This is

the case for almost all the instructions other than not almost for all the instruction other than ret.

So, beq bgt and bn call the target is embedded inside the instructions. So, we can add the pc and the offset to get the branch target; however, the written instruction is special. So, in this case the address where we want to branch 2 is actually there in the inside the register called the written address register which has to be ret from the register file. So, we needed and we are calling that value op 1. So, we need to choose between them with a multiplexer based on the isret signal and field branch pc is generated which is a 32 bits address and the branch pc again goes to the IF stage. So, if you see the IF stage you can see that there are 2 inputs which are coming from the EX stage one is the branch pc and the one isBranchTaken.

So, essentially we are making a choice between pc plus 4 which is the default next pc and the field called branch pc which is essentially target of the branch for a taken branch. We are choosing them depending upon this particular signal and this is telling us either choose the default next program counter or choose branch pc. And branch pc is being generated in the EX stage using this multiplexer as they shown based on the values of the isret signal and data computed from the previous then OF stage.

(Refer Slide Time: 52:19)

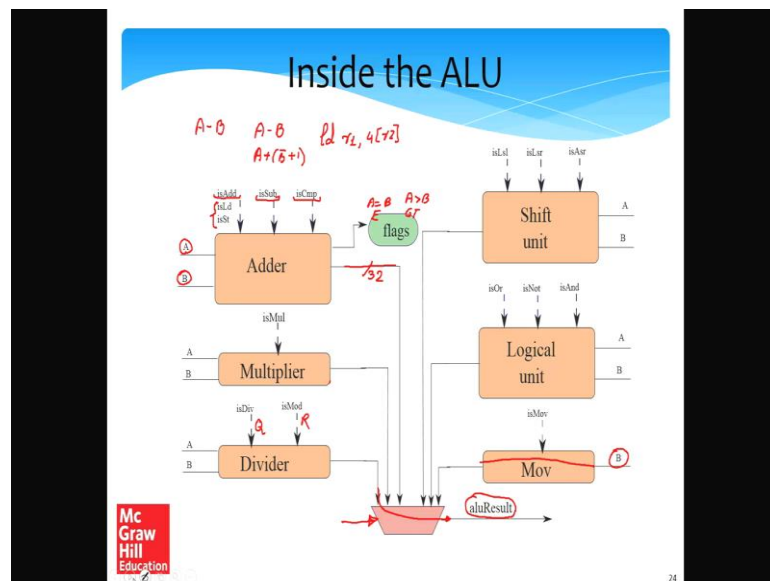


So, now let us take a look at the other component of the execute stage the EX stage called the ALU, the arithmetic logic as unit. So, basically it as 2 inputs let us call them A

and B the first input is an op 1 which is the first source the second input can either be a register which is essentially op 2 the second output of the register file or the immediate immex which is the 32 bits extended immediate. So, we need to choose between them either for the second source either a register or an immediate.

This is done with the help of the is immediate signal which is again generated by the control unit to choose between a register input and an immediate input one of them are chosen, and so that is used as an input to the ALU arithmetic logic unit. So, the arithmetic logic unit can do any kind of an arithmetic logical function such as add subtract multiply divide and even and or a not all kinds of arithmetic and logical functions. So, there are set of ALU signals that direct the ALU and what needs to be done again these are generated by the control unit and ALU produces the result called a ALU result.

(Refer Slide Time: 53:43)



So, let us take look at an inside the ALU what is happening, say inside the ALU we have many functional units. So, what is it exactly that we have? So, the most important unit inside the ALU is the adder. So, the adder actually does several things. So, the adder of course, does regular additions. So, the inputs each of these units are actually the inputs to the ALU which are A and B. So, the A and B were mentioned on the previous slide. So, here we will refer to these 2 inputs as A and B. So, essentially we love giving name to wires. So, we have given a name to the first wire is A and the name to the second is B.

So, the adder also takes 2 inputs A and B. So, then we have a host of control signals generated by the control unit. So, if we have the signal is add as true means that an addition needs to be performed we do the addition and the output comes this wire this wires right. So, wire; so this again contain 30 bits it is not a single wire. So, mind you in 2s compliment in the 2s compliment world addition is the same as subtraction or subtraction is the same I am sorry subtraction is the same as addition. So, to essentially subtract B from A in order to compute A minus B we essentially need to add A plus the 2s compliment of B which is basically the ones compliment of B plus 1. So, any subtraction problem is converted to an addition problem. So, we will use the same adder for it as bit with a little bit of the extra circuitry.

So, the adder can manage that. So, we will have a is sub control signal to subtract 2 numbers then we will have an. So, this subtraction and comparison is a same thing in the sense that when we are comparing A and B what we do is that we subtract B from A and based on the sign of the result we set the flags. So, is compare is the same as a subtract it is just a little set the flags. So, the flags will have 2 bits. So, if A is equal to B we will essentially set bit e to 1 and if A is greater than B then we will set the bit gt to 1 otherwise the both the bits default state of both the bits is 0. So, that is the way that we can compare and comparison subtraction is a same thing the only addition with comparison is that the flags in the flags register get set, also loads and stores use the added and that is basically to add the offset to the base register.

For example, we have a function of this type load r 1 4 r 2. So, in this case the memory address is essentially the contents of r 2 plus 4. So, that is exactly what is done over here in the adder to compute the address of loads and stores. In a similar manner the multiplier works, but it does not do many other things; so which we just have a single control signal called ismal, then we have a divider. So, divider can in principle do 2 things it can either divide A divided by B or it can find the module over the remainder of A divided by B what is it is remainder. So, the quotient can be found out or the remainder can be found out.

So, it depends on the control signal if it is again depending on the type of the instruction. So, this is also one the possible outputs then we have a shift unit which can shift the value. So, here again A and B, A again denotes the particular register that needs that

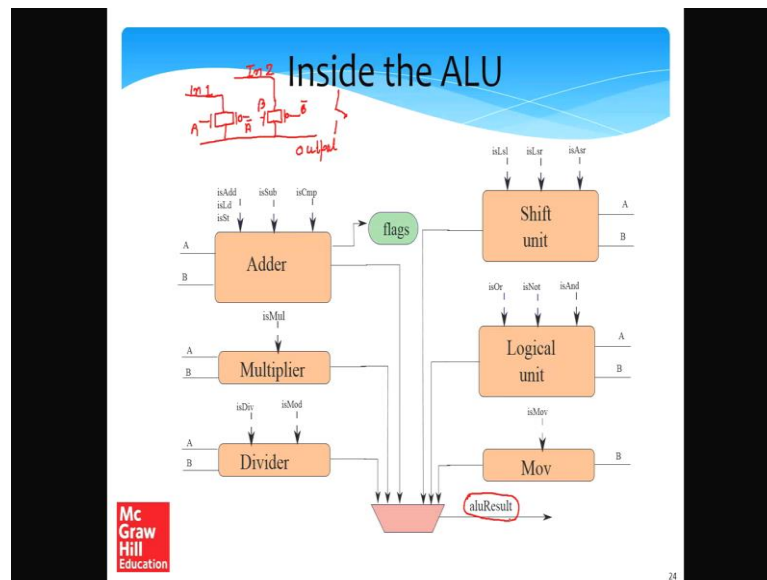
contains a value and B contains a shift amount. So, we can have shifted either logical left shift or a logical right shift or an arithmetic right shift.

So, they have their corresponding control signals. Then we have a logical unit which computes. So, the results of the or not and functions; so not of course, we will not have a second operand, but other 2 or and will have 2 operands. So, the logical unit does that. And then we have a mov unit which is actually very simple. So, what we do is that essentially we do nothing. So, we have a single input that comes in. So, mind you this is input B and the reason that this is input B is because we might move either or register or in immediate into an another register. So, that is the reason it is input B.

So, all that the move unit does is just. So, basically the unit acts as a pass through whether value B simply passes through the unit and goes to the output. So, at the end we have the multiplexer which pretty much gets a lot of inputs. So, in these cases it is getting 6 inputs again a based on the control signals. So, basically this has not been shown because there are been too many arrows, but based on the control signals from the control unit, we need to choose one of them and that is becomes the ALU result. So there is 1 you know nice trick wire which you actually do not need a multiplexer it is called a transmission gate. So, the idea here is that let us say. So, what we want to do over here is that we basically want to. So one of these lines contains valid data the rest contain junk data. So, we want to choose the line let us say may be this line that contains valid data to become the output.

So, in that case simple transmission gate that way that it works actually let me remove the ink on the slide and so transmission gate is a good idea, but it is not the only idea of how to implement such a multiplexer.

(Refer Slide Time: 60:18)



So, let us assume there are 2 inputs of this type. So, let us consider let us call this input 1 and let us again call this as a input 2. So, let us have 2 transistors back to back and let this will be the output line. And so let us the signal that is controlling this be A and let this be the compliment of A which means in both are enabled.

Similarly, let this be B and compliment of B. So, this structure over here is called transmission gate. So now, if you recall discussion from chapter 6; let say that you know A is 1 say if A is 1 this transistor the n mos transistor is on as well as a bar is 0. So, the p mos transistor is on. So, essentially this input is connected to the output. So, the output will take up whatever voltage is there in this input. So, all that we need to ensure is that A and B are not 1 simultaneously. And that can be ensured by not you know by the control unit pretty much do not make A and B on simultaneously.

So, assume that you know these a this a transmission gate is on in the sense the switch is closed, then input one is a same as A output and if B is off then pretty much input 2 in kept floating. So, it will have it will be absolutely disconnected from the outputs. So, you can think of it this is a switch the switch you know absolutely not connected, and similarly we can contain a connect 10 other inputs to the output wire, and as long as we ensure that one of the transmission gates which is essentially just a pair of n mos single pair of n mos p mos transistors. If one of them is enabled meaning transmitting current

conducting then pretty much the output will be equal to that input and the rest of the inputs will be floating, floating means disconnected with the output.

So, that is an easy way of making you know high input multiplexer. It does have its own problems, but classic text on electrical engineering has a better source to take a look at transmission gates, but pretty much you know using transmission gates we can implement a multiplexer of this sort, and what we want at a high level is that we want the output of the ALU called ALU result to be computed correctly.

(Refer Slide Time: 63:15)

The slide is titled "Disabling some Inputs" and contains the following text:

- * We do not want all the **units** of the ALU to be **active** at the same time because we want to save **power**
- * The **instruction** will only use 1 **unit**
- * **Power** is dissipated when the **inputs** or **outputs** make a **transition** ($0 \rightarrow 1$, $1 \rightarrow 0$)
- * We shall avoid a **transition** by not letting the new **inputs** to propagate to **units** that do not require them
- * They will thus have the **old inputs** (**no switching**)

The slide also features the McGraw Hill Education logo in the bottom left corner and a small red icon of a square with a minus sign in the bottom right corner. The number 25 is visible in the bottom right corner of the slide area.

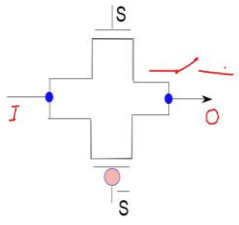
So, this is pretty much talks about the same thing also this slide talks about power consumption. So, the idea is that we do not want all the units of the ALU to be active at the same time, because that what you know consume a lot of power. Because an on the instruction one also will use only one unit a single unit. So, when is power dissipated? So, power is dissipated pretty much when the inputs or outputs make a transition, from a 0 to 1 and 1 to 0. So, why is this case? Because only when there is a transition there is a flow of current. And whenever there is a flow of current there are resistive losses in the circuit and essentially heat gets dissipated.

So, what we need to do is that we can avoid a transition by not letting the new inputs propagate to units that do not require them. So, in a certain sense if a unit consider may be a multiplier that is not being used. So, if let us say inputs are not changed in the sense in a just kept floating in a disconnected from the inputs to the ALU. So, they will

maintain the previous voltages and the output will remain the same. So, as since there is no transition we will not lose any power. And because there is no switching; switching basically means a change in logic levels. So, since there is no switching, there will be no wastage or consumption in power.

(Refer Slide Time: 64:53)

Use a Transmission Gate



- * output = input (if $S = 1$)
- * Otherwise, the **output** is totally disconnected from the **input**

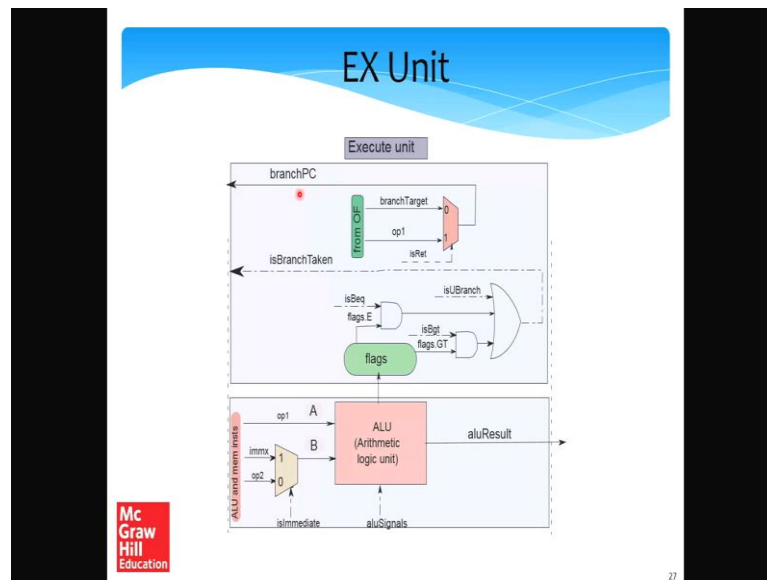
Mc
Graw
Hill
Education

26

So, here again we can use a transmission gate which is exactly a same thing as we have described earlier.

So, the transmission gate pretty much contains a pair of gates in n mos and a p mos. So, the output is equal to the input if s is equal to 1 otherwise the output you know if this is the output and this is the input the output is totally disconnected from the input. So, you can think of it as an open switch. So, in this case also transmission gates can be used to isolate the inputs of the multiplier from the inputs of the ALU.

(Refer Slide Time: 65:29)

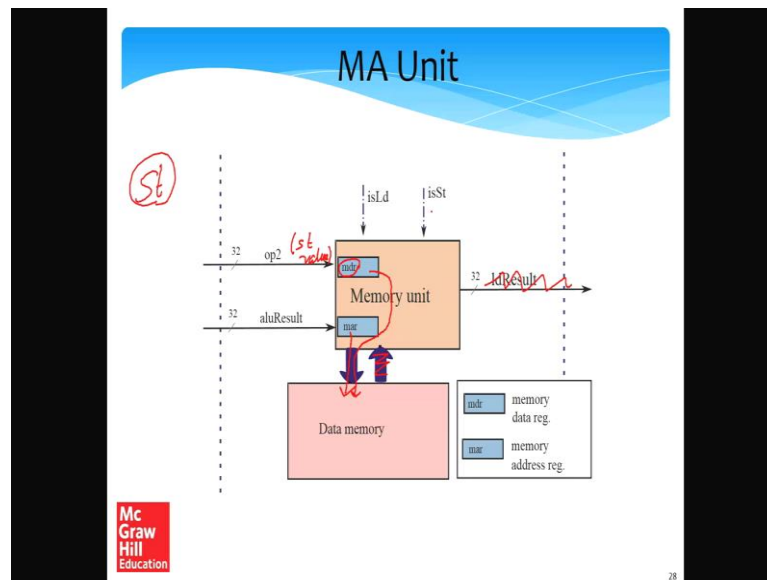


So, given that we can use transmission gates for 2 purposes, one is to isolate the inputs of it is specific sub unit inside the ALU from the overall inputs of the ALU to save power and second we can build them large multiplexer with the help of transmission gates it is its do able.

So, let us summarize our discussion up till now. So, when the execute unit the EX stage contains the arithmetic and a logical unit the ALU. So, the ALU takes in the first operand is the first register source, and the second operand can either be a register or an immediate control whether e is an immediate. Signal based on the ALU signals in the appropriate operation is done. So, the ALU also sets the flags and so the branch unit reads the flags and depending upon the type of instruction whether it is a conditional branch or whether it is an unconditional branch it computes.

If it is necessary to actually change the program counter other than the default. So, that is the `isBranchTaken` signal which says if the current instruction is taken branch or not. Simultaneously you also compute the branch pc which is the final branch target. So, that can either be the written address register which have been ret from the register file or it can be the pc plus the offset embedded in the instruction call the branch target. So, in the any case we use a multiplexer to choose between them and the branch pc goes to the fetch unit.

(Refer Slide Time: 67:19)



So, for the next cycle; let us now consider the 4th stage which is the MA unit on the memory access unit. So, the memory access unit is very simple it as a memory unit which is connected to the data memory inside the memory unit. So let us first look at the results you know; what is it that it requires. So, what we require is that we need the source. So for loads and stores we actually required different thing. So, let us may be first look at the loads. So, let me erase the ink on the slide. See in the case of a load what we need is the address. So, the address is computed by the ALU and the address is the contents of the base register plus the offset which is specified in the immediate. So, this is computed inside the ALU and this comes in and it is stores inside the small register sub register called mar memory address register. So, this is the address in a case of a load.

So, we are discussing the case of a load now. So, then the address flows to the data memory. The data memory fetches the value and it again flows out and it flows out to the next stage and this is called the load result. So, the load result; the only input that the memory unit requires in the case of a load is actually the address is actually the memory address of the load and after that. So, this is in the ALU result field in the ALU result wire this flows to data memory and the outputs comes out.

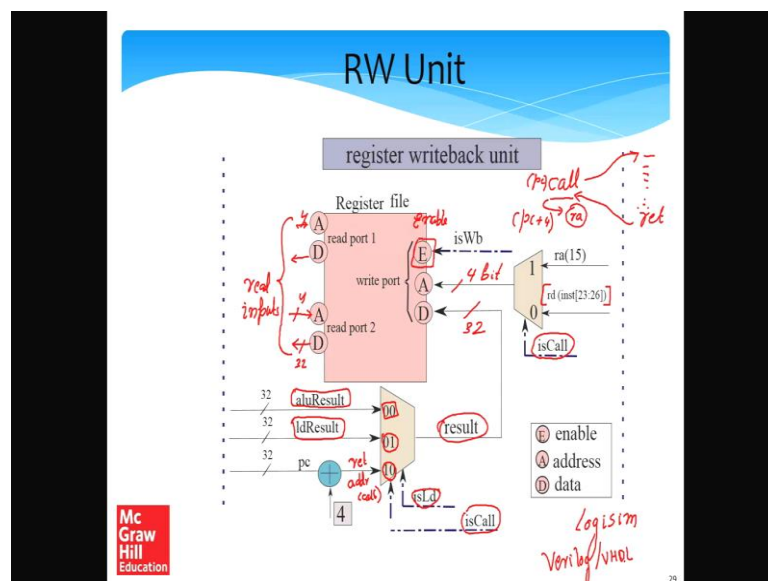
Now let us considered the case of a store; in the case of a store what we shall do is well the first thing that we shall do is we shall erase all the ink on the slide, and the write a big

store over here. So, the store requires 2 things. So, the store requires value to be stored. So, the value to be stored actually, let us go back let us go back to this point where the register that contains the value to be store is the rd field in the instruction bits 23 26. So, if the instruction is store in this value will flow to the register file and the output will be op 2.

So, basically op 2 contains the value that needs to be stored. So, that is this is the store value. And the memory address is there in the ALU result field. So, the ALU result field that contains the address. So, this ALU result contains the address for both loads as well as stores. So, in this case the address gets stored in a mar register inside the memory unit which is not visible outsides. So, it is just the staging area, and the data that needs to be store is stored in mdr register which again is the staging area. So, the address flows to the data memory, and the data that needs to be stored closing to the data memory.

So, inside the memory it is store and we get nothing as output. So, basically store memory the store instruction does not produce any output at all and the job of the instruction at this stage is over. So, the memory unit is very simple in the case of a load it takes an address written a value in the case of a store it takes a value as well as an address and performs the operation. There are 2 control signals that control operation of the data memory isLoad and isstore they are again generated by the control unit.

(Refer Slide Time: 71:30)



Now, let us take a look at the register write units it also known as the write back unit. So, this is slightly more complicated. So, let us take a look at a register file. So, inside the register file we have had seen an inputs, but we are not. So, we are essentially see the read inputs, but we will also have to write in to the register file we are not taken a look at that, but we will look at it now. So, basically level 2 read port a port is an inter face. So, there is a 2 read ports read port 1 and read port 2. So, read port 1 took an address and it essentially gives out the data.

So, similarly lead port one took in address and gave out the data. So, the register file also needs a write port where we will write in something the value of the value that was computed needs to be written into the register file. So, in this case we need to be actually need 3 inputs the first input is the enable input. So, the enable input means that we want to write something. So, enable was not requiring in the case of a read because what is the worst that can be happen in the case of a read, we will read junk which is fine we not use the junk later, but in the case of a write we cannot really afford to write junk right. So, that is not possible it is not advisable and this is not also correct.

So, we need a iswb is write back signal from the control unit which you tell us whether we need to write into register file or not and there are some instructions where we write into the register file such as add subtract multiply divide. There are many instructions in which; however, we do not write to the register file for example, is store that we saw in the last slide. Or a branch instruction we do not write into the register file. So, that is the reason we know the write back signal is important. Next we need to consider the address. So, address basically means which register we want to write to inside a memory address. So, the registers address and since SimpleRisc has 16 registers register address is 4 bits. So, we have choice here.

So, we can choose between several things. So, one is that we take the default register destination which is rd the rd field in the instruction bits 23 to 26. So, that can be that is the default. The call instructions are very special in the since the call instruction what it does is that we jump to a function execute all the things in the function and we call the ret instruction and the ret instruction will come back to the instruction just after the call. So, what we essentially do is that when we call a function we are supposed to write the address of the written address into the ra register or 15.

So, the call instruction is associated with the register write because the written address which is the next instruction after the call each address needs to be written into register r15 or ra. So, thus the register write is involved. Hence we have the isCall control signal generated by the control unit. We choose between r15 the written address register right which chooses between 15 which is the id of the written address register and the default register destination field which is bits 23 to 26 inside the instruction.

So, we choose between them and this becomes the id the address of the register right. So, it is a 4 bit quantity. So, all of this all addresses are 4 bit quantities. So, this is also 4 bits this is also 4 bits. Now all outputs are 32 bits quantities. Next, since we are writing we need to write the result; you know what is the result that we want to write in to? So, we need to write in 32 bits this is slightly complicated not all that hard. So, let us see. So, we need a multiplexer, but what are we choosing from. So, we are choosing first between the ALU result, which is the result that the ALU is producing and this will be the case for add subtract multiply divide you know all arithmetic and logical instructions the result will be the ALU result field. So, that is 1.

The other is the load result which is pretty much the value of a load instruction that you read from memory right, in a previous stage the mem stage the value read from the memory or in the case of a call instruction right. So, consider the call instruction if its program counter is pc, and then we need to have pc plus 4 as the written address. And the written address is which is pc plus 4 which is computed at this point is essentially the value that needs to be written to the register file for the call instruction right.

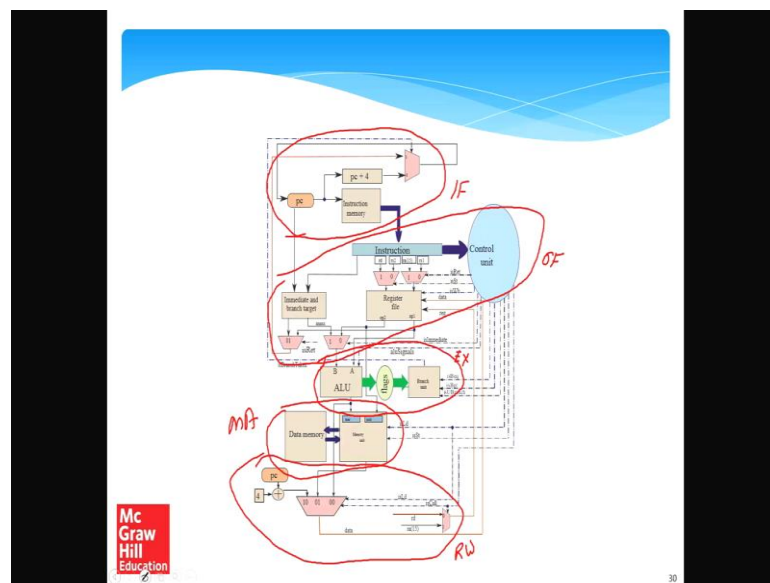
So, you can call this as basically the written address; the written address for the call instruction. We basically have 3 choices what are the 3 choices again. The first choice is the results produced by the ALU right the ALU result. The second choice is the value ret from memory in the case of a load called load result. The third choice is a current program counter plus 4 which is the written address in the case of call instruction, there are 3 choices we need 2 bits we choose between them.

So, let us use 2 control signals isLoad and isCall, call if both of them are 0 we choose the ALU result, if isLoad is true and isCall is 0 means it is a load. So, we choose the load result if isCall is 0 means that it is a function call and is sorry isCall is 1, which means that it is a function call and isLoad is 0 we choose the third input. And after choosing the

value is called result, and that is what is written into the register file. So, this pretty much takes care of you know this discussion of wires that we have been having for the last one hour pretty much covers all the 5 stages of the pipeline in the fair amount of detail.

So, the detail is enough to actually implement this in actual hardware using a hardware description language. So, what I would suggest is that readers can take a look at Logisim. So, Logisim is freely available on the web with an open source licence. So, readers can take look at Logisim because it is very easy to use especially for first timers. So, they can design this circuit on Logisim and so basically this circuit will work in you know as you shown it is essentially everything that we show as to just to be you know enter into Logisim and Logisim is good support for drag and drop elements. So, it is possible to take design multiplexers and so on.

(Refer Slide Time: 79:59)



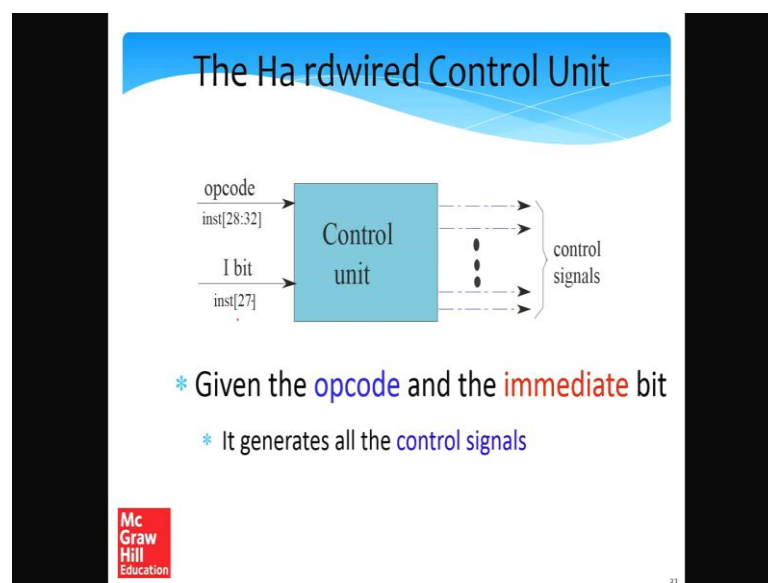
That is 1 option and so those who know hardware description languages such as very log and VHDL, but also implemented you will find a Logisim model of this processor and a very log VHDL model of this processor on the course website and so that can be a good starting point VHDL model.

So, now it is time to summarize whatever we have done. So, this is pretty much our design as up till now. So, the top parts; so I will not be going to the details it is not required we have discussed all of this. So, this is the fetch unit. So, I just taken all the individual diagrams made one large diagram that is all nothing more. So, this part is the

of unit right along with the control part. So, if the control unit is thought of as a black box which generates you know all of these signals to control all the multiplexers. So, you see all multiplexers are controlled by these dotted lines generated by the control unit. Then this part is the EX stage, and this part is a memory access stage the ms stage and finally, this part over here which in the sense all of it is data back to the register file is the RW stage.

So, we have discussed all of this and the, but this is just showing you the entire processor let me remove all my annotations and again go back to this beautiful design. So, this is a processor for you very much and this is strong enough potent enough capable enough to run the entire SimpleRisc entire SimpleRisc ISA. And so tells absolutely no problem and you will find models of this implemented Logisim and VHDL on the website. So, readers are more than welcome to try it out.

(Refer Slide Time: 81:58)



Let us now look at the design of the control unit. So, the control unit is very simple it is a piece of logic the inputs to the control unit of the opcode the 5 bit opcode. So, it bits 20 8 to 32 in the instruction, and the I bit the immediate bit which is bit number 27 the outputs are all the control signals that we would require. So, basically given the 6 bit input the opcode 5 bits and one bit immediate bit it generates all the control signals.

(Refer Slide Time: 82:28)

SerialNo.	Signal	Condition
1	<i>isSt</i>	Instruction: <i>st</i>
2	<i>isLd</i>	Instruction: <i>ld</i>
3	<i>isBeq</i>	Instruction: <i>beq</i>
4	<i>isBgt</i>	Instruction: <i>bgt</i>
5	<i>isRet</i>	Instruction: <i>ret</i>
6	<i>isImmediate</i>	<i>I</i> bit set to 1
7	<i>isWb</i>	Instructions: <i>add, sub, mul, div, mod, and, or, not, mov, ld, lsl, lsr, asr, call</i>
8	[<i>isUBranch</i>]	Instructions: <i>b, call, ret</i>
9	<i>isCall</i>	Instructions: <i>call</i>

So, let us basically look at the conditions of these control signals. So, this list contains all the control signals that we have seen in the in the past few slides. So, we are seen a isstore. So, the conditions with the instruction is a store similarly isLoad when instruction load is beq bgt ret is immediate is set to 1. In the I bit instruction is set to one the difficult one is iswrite back. So, iswrite back is set for those instructions which write to a register.

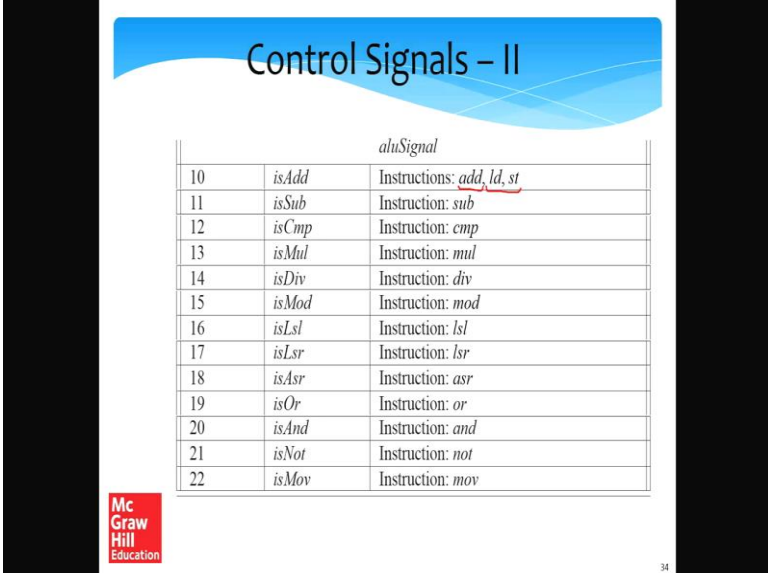
So, first the instruction that is side write to a register are arithmetic instructions, add subtract multiply divide and modulo. Then their logical instructions and or a not, we then have the move instructions which moves an immediate or a register to another register out of the 2 memory instructions load and store only one of them writes to a register it is a load instruction stored does not write to a register. Then we have the 3 shift instructions that write to a register, and we have already discussed the case of the call instruction which writes to a register. So, this is mind you implicit in the sense that when we call function the written address is written to the written address register.

We have discussed the case of isubbranch is unconditional branch. So, definitely the b instruction will be in that. So, b is for branch. So that will be covered. And this condition is also true when we have the call and ret instructions. So, call and ret instructions are also unconditional branches. So, that is reason this instruction is also I mean this

instruction also comes in that. And so as we have discussed in this slides that when. So, let me go back to the execute slide I will promise that we will go back to it

So, when we are discussing the branch unit we have talk about the isubbranch signal. So, isubbranch signal happens when we are guaranteed that we will take the branch and that is possible for the b instruction and the call and ret instructions. Because it is guaranteed that the value of the program counter will change. Then the isCall instructions signal is set when the when the instruction is call. Let us now take a look at. So, those were the basic control unit signals for entire data path.

(Refer Slide Time: 85:23)



The slide is titled "Control Signals - II" and features a table with the following data:

<i>aluSignal</i>		
10	<i>isAdd</i>	Instructions: <u>add</u> , <u>ld</u> , <u>st</u>
11	<i>isSub</i>	Instruction: <i>sub</i>
12	<i>isCmp</i>	Instruction: <i>cmp</i>
13	<i>isMul</i>	Instruction: <i>mul</i>
14	<i>isDiv</i>	Instruction: <i>div</i>
15	<i>isMod</i>	Instruction: <i>mod</i>
16	<i>isLsl</i>	Instruction: <i>lsl</i>
17	<i>isLsr</i>	Instruction: <i>lsr</i>
18	<i>isAsr</i>	Instruction: <i>asr</i>
19	<i>isOr</i>	Instruction: <i>or</i>
20	<i>isAnd</i>	Instruction: <i>and</i>
21	<i>isNot</i>	Instruction: <i>not</i>
22	<i>isMov</i>	Instruction: <i>mov</i>

Mc Graw Hill Education logo is visible in the bottom left corner of the slide.

But then let us look at the ALU signals. So, the signals sent to the ALU. So, we are look at them. So, is add signal is when you want to ALU to do an addition. That holds true for the add instruction as well as the load and store instructions because it is necessary to add the offset to the contents of the base register. Then we have the sub instruction subtract compare instruction multiply divide modulo the 3 shifts or and not and move. So, for each whenever the instruction is any one of them we generate an appropriate signal.

(Refer Slide Time: 86:13)

Control signal Logic

opcode

op ₅	op ₄	op ₃	op ₂	op ₁
-----------------	-----------------	-----------------	-----------------	-----------------

immediate bit

I

Serial No.	Signal	Condition
1	<i>isSt</i>	$\overline{op_5} \cdot op_4 \cdot op_3 \cdot op_2 \cdot op_1$
2	<i>isLd</i>	$op_5 \cdot op_4 \cdot op_3 \cdot op_2 \cdot \overline{op_1}$
3	<i>isBeq</i>	$op_5 \cdot \overline{op_4} \cdot \overline{op_3} \cdot \overline{op_2} \cdot \overline{op_1}$
4	<i>isBgt</i>	$op_5 \cdot op_4 \cdot op_3 \cdot op_2 \cdot op_1$
5	<i>isRet</i>	$op_5 \cdot \overline{op_4} \cdot op_3 \cdot \overline{op_2} \cdot \overline{op_1}$
6	<i>isImmediate</i>	<i>I</i>
7	<i>isWb</i>	$\overline{op_5} + \overline{op_4} \cdot \overline{op_3} \cdot op_1 \cdot (op_4 + \overline{op_2}) + op_5 \cdot op_4 \cdot op_3 \cdot op_2 \cdot op_1$
8	<i>isUbranch</i>	$op_5 \cdot \overline{op_4} \cdot (op_3 \cdot op_2 + op_3 \cdot \overline{op_2} \cdot \overline{op_1})$
9	<i>isCall</i>	$op_5 \cdot \overline{op_4} \cdot \overline{op_3} \cdot op_2 \cdot op_1$

35

So, whatever written on this slide is let us consider the 5 opcode bits let us call op 5 to op 1 and the immediate bit i. So, I just had written down the conditions for finding for essentially for essentially detecting these signals or setting these signals.

So, this condition basically means that if the op 5 bit is 0 and the rest of the bits are 1 0 1 1 1 we have the isstore instruction the istore signal is set to 1. So, similarly there are conditions for the rest of the signals. Since there write back iswrite back signal was complicated it as a slightly longer form. So, it is possible. So, whenever such a circuit is being designed. So, typically the control unit takes time to design, but since these Boolean equations are given, it is very easy to design this control unit in a hardware. So, it can be very easily done on a bit Logisim or any other hardware description language. All that needs to be done with this logic these equations need to be written and implemented in hardware. So, this brings to the end of the chapter of processor design with the hardware control unit.

So, we shall now look in the next part on microprogrammed processors which are a different kind of processors that are far more flexible and they are large part of it can be changed by the programmer, but their performance is somewhat at the lower side.