

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

**Lecture – 24**  
**Processor Design Part-II**

(Refer Slide Time: 00:29)

### Outline

- \* Outline of a Processor
- \* Detailed Design of each Stage
- \* The Control Unit
- \* Microprogrammed Processor
- \* Microassembly Language
- \* The Microcontrol Unit

37

So, we have looked at one kind of a processor known as a hardware processor. So, in this hardware processor the main brain of the processor is the control unit.

(Refer Slide Time: 00:39)

### Control Signal Logic - II

<i>aluSignals</i>		
10	<i>isAdd</i>	$\overline{op5.op4.op3.op2.op1} + \overline{op5.op4.op3.op2}$
11	<i>isSub</i>	$\overline{op5.op4.op3.op2.op1}$
12	<i>isCmp</i>	$\overline{op5.op4.op3.op2.op1}$
13	<i>isMul</i>	$\overline{op5.op4.op3.op2.op1}$
14	<i>isDiv</i>	$\overline{op5.op4.op3.op2.op1}$
15	<i>isMod</i>	$\overline{op5.op4.op3.op2.op1}$
16	<i>isLsl</i>	$\overline{op5.op4.op3.op2.op1}$
17	<i>isLsr</i>	$\overline{op5.op4.op3.op2.op1}$
18	<i>isAsr</i>	$\overline{op5.op4.op3.op2.op1}$
19	<i>isOr</i>	$\overline{op5.op4.op3.op2.op1}$
20	<i>isAnd</i>	$\overline{op5.op4.op3.op2.op1}$
21	<i>isNot</i>	$\overline{op5.op4.op3.op2.op1}$
22	<i>isMov</i>	$\overline{op5.op4.op3.op2.op1}$

36

The control unit as we have just seen generates all the control signals. And these control signals control the movement of data inside the processor right. So, for different kinds of instructions different control signals are generated.

So, this kind of a processor with a hardware control unit is fairly common. In fact, that is the standard, but there is another kind of processor as well that gives us a lot more flexibility. In certain situations, it is used also in modern processor such as an intel line of processors, there are some instructions which run you know essentially the processor itself contains the hardware part and what we call the microprogrammed part where some very complex instructions are implemented using the lantern the microprogrammed part. So, let us take a look a microprogrammed processor in slightly more detail.

(Refer Slide Time: 01:39)

**Microprogramming**

- \* Idea of **microprogramming**
  - \* Expose the elements in a processor to software
  - \* Implement instructions as dedicated software routines  $\{37 \times (r1 + r2)\}$
- \* Why make the **implementation** of instructions flexible ?
  - \* Dynamically change their behaviour
  - \* Fix bugs in implementations
  - \* Implement very complex instructions

McGraw Hill Education

38

So, what is the idea of microprogramming? So, in this case we expose the internal elements of a processor pretty much to software right. So, to the assembly programmers we expose the internal details of the processor which were otherwise hidden. So, this allows us to pretty much code in instruction right. So, consider in the add instructions and add instructions are doing many things. So, what we can do is that we can add a new instruction or modify an add instruction to for example, instead of add 2 numbers add 3 numbers right. So, we can change the behavior of instructions we can do many more things, and we can also add very complicated instructions which are like add register

contains of  $r_1$  and  $r_2$  and multiply them by 37. So, we can you know introduce an instruction of this kind right 37 times  $r_1$  plus  $r_2$ .

So, you know we can definitely introduce instructions of this kind and this. So, we are searching for such kind of a processor which can give us this flexibility of doing many things implementing. So, let me you know walk backwards implementing complex instructions fixing bugs in implementations of current instructions. And also changing the behavior of instructions as I said instead of an add instruction we can replace it with this if you know there is a requirement by applications. So, this is all of this is great this comes at a price and the price is speed of course. So, processors with hardware control units are much faster and processors with microprogramming are relatively slower. So, the price is speed and efficiency.

(Refer Slide Time: 03:34)

The slide is titled "Microprogrammed Data Path" and contains the following text and annotations:

- \* Expose all the state elements to dedicated system software – **firmware**
- \* Write dedicated routines in **firmware** for implementing each instruction
- \* **Basic idea**
  - \* 1 SimpleRisc Instruction → Several micro instructions
  - \* Execute each micro instruction → **add** → **Microprogram**
  - \* We require a microprogram counter, and microinstruction memory

Handwritten annotations include:

- A bracket grouping the first two bullet points.
- A diagram showing a box labeled "FW" (Firmware) with "SW" (Software) above it and "HW" (Hardware) below it.
- A bracket under "Basic idea".
- A bracket under "1 SimpleRisc Instruction → Several micro instructions".
- A bracket under "Execute each micro instruction".
- A bracket under "We require a microprogram counter, and microinstruction memory".
- Handwritten labels "PC" and "IM" in circles.
- A small "McGraw Hill Education" logo in the bottom left corner.
- A small "39" in the bottom right corner.

So, we need to create a microprogrammed data path. Some microprogramming is different from conventional programming because in this case we are writing the code for each instruction. So, that is why it is microprogramming. So, what we do is that we expose all the state elements inside a processor to system software so, so this is also known as firmware where we can write dedicated routines. In firmware which basically means the part of the system software that in a very intricately controls the behavior of hardware. This can be done for implementing each instruction. So, the way that we look

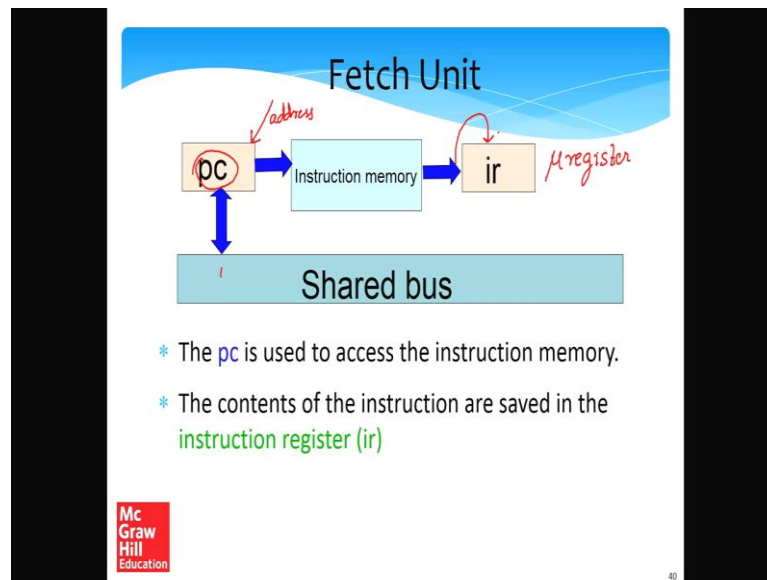
at we have conventional software. We have conventional hardware and we can have a small thin layer in the middle called firmware.

So, firmware is still software, but we still distinguish it from it because unlike traditional software the firmware is loaded or stored in a specific location in hardware and it is used to very intricately control all that the hardware does. So, it has far more visibility into hardware as compared to traditional software so, that is the reason we refer to this layer as firmware and it is a very formal term in the systems community. So, you can have firmware for all kinds of systems. You can have firmware for disks you can have firmware for network cards for routers everything right it is possible to have a firmware layer.

So, firmware is again software, but software meant for a different purpose. So, let me talk about the basic idea of microprogramming. So, in this case we take one simple risk instruction right you know like add subtract multiply and we break into several small micro instructions. We execute each micro instruction pretty much one every cycle and so pretty much what we do is for an add instruction we replace it with a microprogram which can have many micro instructions.

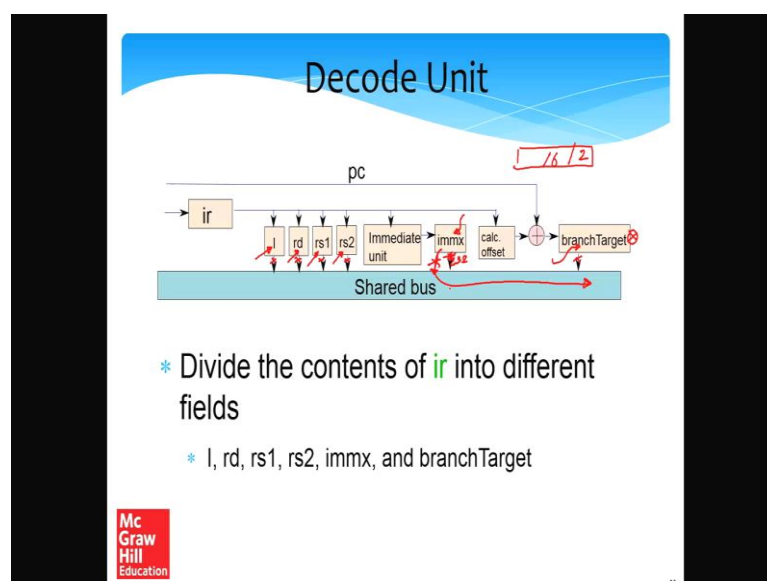
So, what we do is that we start from the top and keep on executing till the bottom and so basically this is this is the small program by itself where instead of one high level risk instruction, we execute several micro instructions once this micrprogram is done say it is typically represented something like this microprogram once this is done it means that we have completed finishing the execution of this instruction. So, we require a microprogram counter and a microinstruction memory, very similar to the pc that we had and the instruction cash the instruction memory sorry I should not be using the word cash here because we have not introduced it, but the instruction memory right that we have been using.

(Refer Slide Time: 06:58)



So, let us now this also have different units. So, let us take a look at the fetch unit. So, in the fetch unit we still have the pc register here. So, the pc register is used to access the instruction memory and load the contents of a regular simple risk instruction the contents of the instruction are saved in the instruction register ir. So, mind you this is the microregister and this this microregister which you are calling ir instruction register contains the contents of the; basically contains the contents of the instruction, whereas address is given in the pc. So, the address of high level simple risk instruction we use this access instruction memory and the contents gets transferred to the ir register.

(Refer Slide Time: 07:50)



Let us now take a look at the decode unit. So, in the decode unit the input pretty much comes from the ir register and so basic. So, so then what little bit of circuit tree does which is pretty much hidden, is that we extract all the fields from the ir register. So, for some instructions some of the fields might not make sense they might contain junk, but nevertheless we extract all the fields for example, we extract the i bit the immediate bit we extract the rd field which in normal cases stores the id of the destination register.

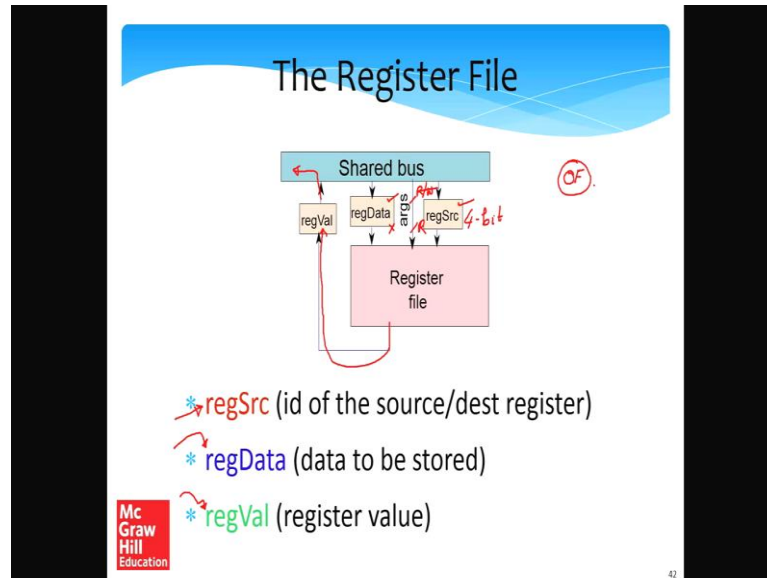
We extract the first source register the second source register we also have an immediate unit that extracts the 32 bit extended immediate after taking modifiers into account. So, it extracts the immediate right. So, basically the 18 bit immediate is extracted to 32 bit immediate. So, recall that you know in immediate in simple risk is actually 16 bits plus 2, where the 2 bits specify the modifier. So, the 32 bits are available on in the immx register.

In addition, what we do is that we take the offset part from the instruction. So, of course, this point we do not know what is the type of the instruction, but we repair for all kinds of instructions, and we pre compute all of these terms write the immediate and also the branch target, but of course, it is not necessary that every instruction is a branch. So, if an instruction is not a branch this microregister will contain junk, but nevertheless in the interest of time and performance it is a good idea to compute things in advance just in case they required. So, to compute the branch target what we do we take extract the offset part from the instruction register add it to the current pc and we get the branch target. So, you also compute this.

All those, these resistors are connected on a shared bus. What is the shared bus? So, shared bus is basically a set of copper wires to which all of these resistors are connected, but it is not the case that all of them are feeding their data to these copper wires right. So, then it will not work. So, by default you know these links between the registers and the bus are actually turned off they are open circuits, but it is possible for us as we shall see in the next few slides to you know occasionally read these registers. So, read this registers or rather microregisters what we need to do is we need to send a command. For example, to read a immx and then this connection from immx to the bus will get enabled, and what will see on the bus is a value of immx which others can read.

After reading again we will send one more command is automatically this link will get cut off and the bus will not contain the value of immx. So, what again is a shared bus it is a set of shared copper wires.

(Refer Slide Time: 11:12)



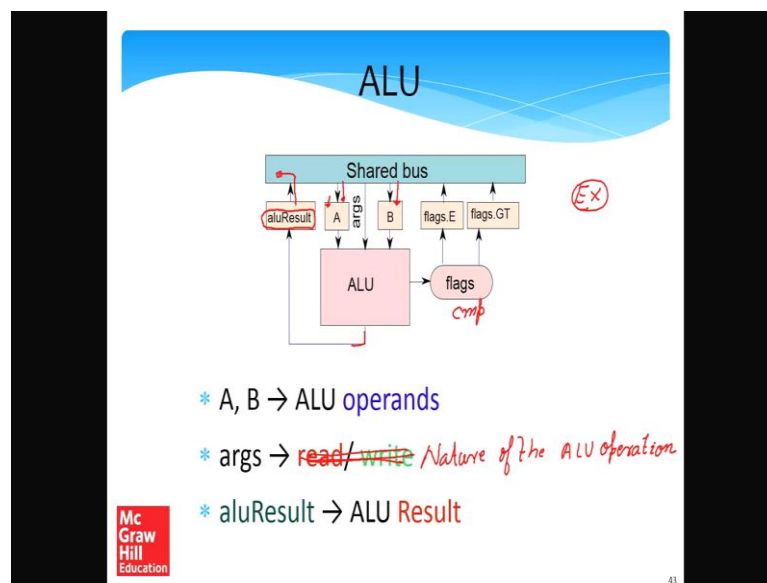
So, now let us take a look at the next stage right the execution stages the EX stage. So, this contains the register file. So, what we shall do. So, we augment this part of the processor with actually 3 more microregisters. So, the registers are as follows. So, one of them is regSrc. So, regSrc is a register that we want to access either for read or write it does not matter, but regSrc contains a 4 bit value mainly because we have 16 registers. And this is the id the register that we want to access. Then we have regData data which means that if you want to write to a register then the value of regData will come from the shared bus, it will be saved in the regData microregisters and from there it will go to the register file.

The args are the arguments basically specify whether we want to do a read in a reader register or write to an actual register. So, mind you the register file here is exactly the same as the previous processor. So, let us may be adopt this convention that whenever I introduce a new structure for the microprogram processor, I will separately specify use the word the microregister as far as possible otherwise you know almost everything remains the same. So, what is regData again? So, when I am writing to a register regData contains the value that needs to be written.

Assume that I want to read a register then the value of regData is relevant will not be that; however, the value of the arguments will be equal to R which means I want to read a register. The id of the register will be there in read source and the contents that are read from the register file will go and get stored in the regVal microregister. Subsequently if some other stage in some other part of the processor wants to know what exactly was read from the register file, then the regVal microregistered can supply the value on the shared bus. So, we have regSrc which contains the id of the register regData are contain something if something has to be written and regVal which is the value of the register which was just read.

Let us take a look at the ALU. One small correction here in this slide I had incorrectly mention that this one the execution stage that is not correct this is still a part of the operand fetch or the OF stage. So, this is one correction that I would like to make.

(Refer Slide Time: 14:12)



So, now we enter the proper EX stage and EX stage the main elephant in the room for the EX stage the ALU. So, controlling the ALU is easy. So, what we have done previously is that we have kept. So, the pink boxes here are things are artefacts of or earlier in a hardware processor design and these are all the new elements. So, basically this light brown colour these are all the new elements that we add. So, let us take a look at them. So, let us assume you want to perform an ALU operation.



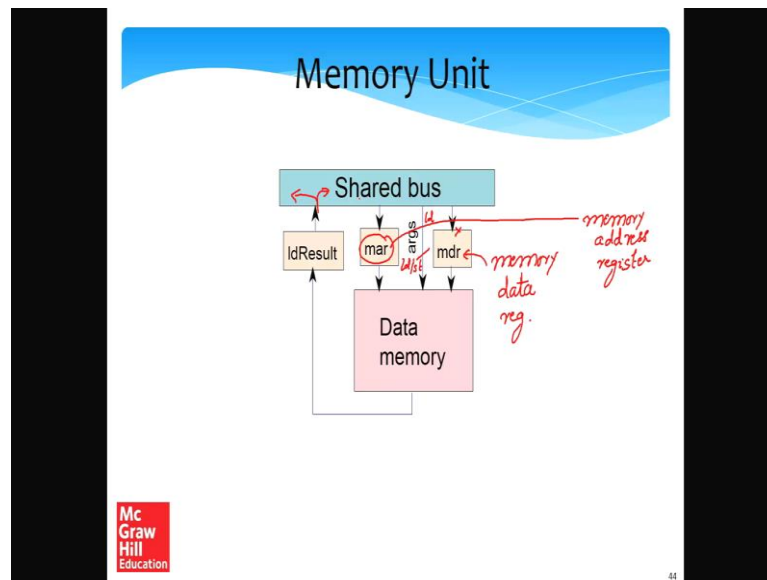
If you want to perform an ALU operation then we will have 2 operands right in a most of the time we will have 2 operands like add multiply subtract sometimes we will have one we will take a look into that, but most of the time we will have 2 operands. So, we will add one 32 bit microregister called A which stores the first operand, we will have one more 32 bit microregister called B which will store the second operand right that we will read from the shared bus. So, the second operand will be stored in microregister B. Args which is incorrectly specified here will contain the nature of the operation the ALU needs to perform. So, I should rather write nature of the ALU operation you know add subtract multiply what kind.

So, args will contain a nature what exactly you know you want the ALU to do. So, that will be there in args and all of them will be read from the shared bus. So, it is not that you know is all of these units are active all the time, but we will have a way activating only in a certain subset of these units depending upon what needs to be computed.

So, let us assume that the ALU is active. So, it will read the contents of args from directly from the bus A and B after that the result of the ALU after the computation is done towards end of the cycle will be stored in another microregister called ALU result, and for each of these microregisters they will contain their value till their over written. So, aluResult can further supply that this aluResult microregister can supply the value to others in a unit on demand of course, via the shared bus everybody operates via the shared bus.

Similarly, we will have a flags register that we have discussed in our previous lecture. So, which stores the flag of a compare and we will also expose the flags. So, we will have 2 small sub microregisters flags dot E and flags dot GT e for equality and d t for greater than these 2 sub microregisters will also be able to export their values on the shared bus to whoever who requires it right. So, these are all the microregisters I am sorry that we add in the ALU stage A and B for the inputs aluResult contains the output flags dot E and flags dot GT contain the results of the flags. So, recall that is only the cmp instruction that changes them otherwise the flag values reflect the state of the flags that was set by the last cmp compare instruction.

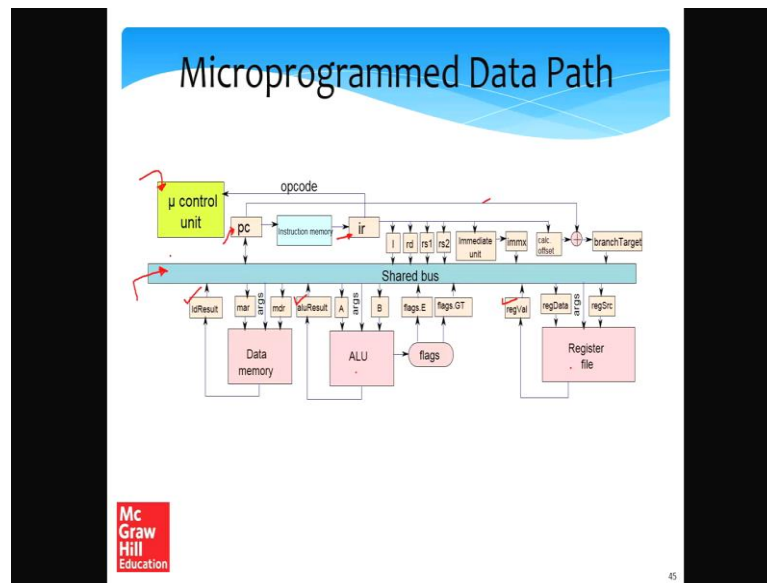
(Refer Slide Time: 17:58)



Now, let us go to the MA stage phase the memory unit the memory unit does something very similar we add you know following the same in a line of thought we add microregisters to the data memory. So, we add a memory data register mdr. Similarly, we have a memory address register. The value of the args say whether the operation is a load or a store right which operation it exactly is. So, whenever we are doing a store operation we will send the memory address via the memory address microregister, and the data that needs to be stored will come via the shared bus to the memory data register. Subsequently and also the type of the operation which will be a store this will be this will go to the data memory, where the store will be affected. If the operation is a load then of course, the args here will contain the fact that this is a load operation.

And so the mdr will not have any value, but the load will have an address. So, the address will be stored in the mar or memory address register and the result of the load will be stored in another microregister called ldResult, ldResult this is similar to l u result exactly similar in spirit. So, this will contain the result of the load which can be given to others on the shared bus. So, what we have done is we have taken again our old unit which is the data memory in thing and we have added 3 new microregisters mar for the address mdr for the store data and ldResult for the data that we read from data memory. So, we add these 3 microregisters.

(Refer Slide Time: 20:16)



So, what is our micro program data path look like now? This is what it looks like have what I have done is of taken all the small diagrams that we have been introducing over the last few slides, and I have collated them into one large diagram. So, let me starts from one side; so given the pc or the program counter. So, this is connected to the shared bus of course, and so this is not maybe it was not mentioned in that slide, but it needs to be connected to the shared bus both for read and write it access the instruction memory the contents of the instruction memory go to the ir instruction register which contains the contents of the instruction we have a little bit of circuitry that ba breaks of the instruction in to his constituent and fields namely the immediate bit rd rs 1, rs fields the extended immediate the 32 bit immediate that we get from the 18 bits embedded in the instruction.

We also calculate the offset you know there will be an offset some bits in the instruction specify an offsets we added to the pc to get the branch target. Simultaneously the instruction particularly is the opcode of the instruction goes to the micro control unit which decides in a for one given instruction there is for every separate instruction there is a separate micro program. So, it decides which micro program needs to be run and based on that it run some separate micro program for each instruction. So, what again is the advantage well the advantage is we have complete flexibility as we shall just see. So, we can add new instructions you can change the behavior of existing instructions and all of this can be done at the level of firmware which is very specialized software. For

example, a processor is sold and it is a part of a system after that we can change the way it is our instruction operates.


So, that is pretty cool in a lot of cases. So, it will do that what we need is that we need to consider some of our main functional units that were there in our previous processor. For example, the register file ALU and data memory for each of these we add microregisters between the unit and the shared bus. So, the microregisters buffer the inputs and also store the outputs. And outputs will remain in that state till the unit is access once again and the inputs of the outputs are over written. And to control the units we have some bits that come from the shared bus namely the args arguments bits 2 for example, tell the register file to read or write or to tell the ALU what kind of operation it needs to do or to tell the data memory whether it is a load or a store.

Once the operation is complete the results will all be there in these result micro resistors ldResult aluResult and regVal which can then be use by other units if required. So, we shall see write micro programs we will see; how all of this works. And the crux of the entire processor is a micro control unit which runs the microprogram for each high level simple risk instruction and the shared bus which in a conceptually is just the set of copper wires. Now that we have seen the microprogrammed processor, let us look at micro assembly language.

(Refer Slide Time: 24:06)

### Internal Registers

SerialNo.	Register	Size (bits)	Function
1	pc	32	program counter
2	ir	32	instruction register
3	l	1	immediate bit in the instruction
4	rd	4	destination register id
5	rs 1	4	id of the first source register
6	rs 2	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


47

So, before we begin let us look at all the registers that we have introduced. So, we have taken the pc register from our previous implementation of a processor and this is the typical program counter; so no changes in that.

We have introduced the ir register to contain the contents of the instruction. The i instructions is a single bit I am sorry the i microregister which is single bit the immediate bit rd rs 1 and r rs 2 which contain the destination register id, id at the first and second sources the 32 bit extend and immediate stored in the immx microregister. The branch target in the branch target microregister regSrc regData.

(Refer Slide Time: 24:55)

**Internal Registers - II**

11	<i>regVal</i>	32	value read from the register file
12	<i>A</i>	32	first operand of the ALU
13	<i>B</i>	32	second operand of the ALU
14	<i>flags.(E)</i>	1	the equality flag
15	<i>flags.(GT)</i>	1	the greater than flag
16	<i>aluResult</i>	32	the ALU result
17	<i>mar</i>	32	memory address register
18	<i>mdr</i>	32	memory data register
19	<i>ldResult</i>	32	the value loaded from memory

Mc  
Graw  
Hill  
Education

And regVal which encapsulate the register file. Similarly, A B flags dot E flags dot GT and aluResult which encapsulate the ALU and finally, the mar mdr and ldResult macro registers which encapsulate the data memory.

(Refer Slide Time: 25:23)

**Microinstructions**

**Basic Instructions**

- \* **mloadIR** → Loads the instruction register (ir) with the contents of the instruction.
- \* **mdecode** → Waits for 1 cycle. Meanwhile, all the decode registers get populated  
*I, rd, rs1, rs2, immx, branchTarget*
- \* **mswitch** → Loads the set of micro instructions corresponding to a program instruction.  
*Simple Risc*

McGraw Hill Education

49

So, let us introduce some basic instructions we do not want to introduce many, but let us at least introduce few right as many as a required.

So, the basic instructions or other basic micro instructions are so, let us make a convention that we will start all micro instructions with letter m. So, let us look in mloadIR. So, mloadIR loads the instruction register ir with the contents of the instruction. So, whatever are the contents of the instruction, they come from the instruction memory and get stored in the instruction register ir, fine. So, this is mloadIR. Next is mdecode this is a one cycle instruction. So, all the decode registers get populated. So, we read the contents of the instruction from ir and all the decode registers which are essentially the immediate bit ir d r1. So, recall that we I am sorry we are interchangeably using the term microregister and register. So, please infer them from the context.

So, we our decode macro register is rs 1 ir d rs 1 rs 2 immx and branch target, these are our decode registers microregisters. So, they what we do is that the mdecode instruction we wait for one cycle and in the one cycle we expect that the circuitry that takes the contents of the instructions from ir would have populated each of these registers here right. Subsequently once we have decoded we call the mswitch micro instruction which loads the set of micro ins instructions corresponding to a given program instruction a simple risk instruction. Let us for example, if let say the program instruction is add we will know that it is an add in the decode stage.

So, this will go to the micro controller and this will load the set of micro instructions in to some internal storage such that for an instruction for an add instruction we can execute all the micro instructions are correspond to it.

(Refer Slide Time: 28:09)

Move Microinstructions

- \* `mmov r1, r2 : r1 ← r2`
- \* `mmov r1, r2, <args> : r1 ← r2, send the value of args on the bus`
- \* `mmovi r1, <imm> : r1 ← imm`

Mc Graw Hill Education

50

So, this will happen with the mswitch instruction. Now let us also take look at 2 more instructions which are very basic. So, all the instructions in a micro program by plan actually the extremely basic. So, we have taking a look at mloadIR which essentially loads the contents from the instruction cash to ir register mdecode which simply does a decode. And mswitch which for a given simple risk instruction loads it is corresponding micro program. So, mmov and mmovi are straightly different. So, m in mmov given 2 registers r1 and r 2 it simply takes the contents of r 2 puts it in r1 we have another variant of mmov where it additionally takes a few bits called the arguments.

So, what it does is that it puts r 2 and r1 that it does and also sends the value of args on the bus. So, basically it has these 2 if an additional arguments set of arguments are present then, the value of the args field is sent on the bus on the shared bus that is. So, this is what move does it comes in 2 formats one with arguments one without arguments. And we have mmovi say mmovi is the mmov immediate where given in immediate the immediate is moved in to the register r1. Where is the first register? Whatever it is, but mmovi pretty much moves the contents of the immediate in to the register.

(Refer Slide Time: 29:48)

**Add and Branch Microinstructions**

- \* **madd**  $r1$ ,  $imm$ ,  $\langle args \rangle$ 
  - \*  $r1 \leftarrow r1 + imm$
  - \* send  $\langle args \rangle$  on the bus
- \* **mbeq**  $r1$ ,  $imm$ ,  $\langle label \rangle$ 
  - \* if ( $r1 == imm$ ),  $\mu pc = addr(label)$
- \* **mb**  $\langle label \rangle$ 
  - \*  $\mu pc = addr(label)$

McGraw Hill Education

51

We have a few more. So, we introduce 3 very basic instructions, which are micro instructions which are required for every instruction namely mloadIR mdecode and mswitch with an introduce 2 genetic move instructions mmov and mmovi. So, one to move a register 1 to move an immediate now le now we need a little bit of add and branch micro instructions. So, one the add micro instruction that we shall introduces a madd. So, it takes the first argument is a register the second argument is an immediate and the third argument is basically a set of bits called the args field. So, what it does is that it sets r1 similar to x86 where you know the des one destination and source their common.

So, we have it in a similar way where r1 is said to the old value of r1 plus the immediate. So, r1 is equal to r1 plus the immediate and we send the value of args on the bus the arguments we send them on the bus. Similarly, we have mbeq is it is a micro branch if there is an equality. So, the equality is like this we specify register a specify an immediate and the label micro label rather see if r1 is equal to equal to immediate which means if r1 is the same as the contents of r1 is the same as the contents of m. We said the micro pc in the micro program to the address of this label. So, this is an artefact of traditional assembly say it works in exactly the same way where is just that we send a register r1 immediate, we compare them with the comparison successful we do a branch.



We also have an unconditional branch `mb` which just takes a label when we said the macro `pc` to the address of the label you know unconditionally. So, we added. So, we introduce 3 new micro instructions on the slide, `madd` for a micro instruction addition `mbeq` which is a conditional micro branch and `mb` which is an unconditional micro branch.

(Refer Slide Time: 32:16)

The slide titled "Summary of Microinstructions" contains a table with 8 rows. The table columns are SerialNo., Microinstruction, and Semantics. Handwritten red annotations include arrows pointing from `pc` to `ir` in row 1, a bracket on the left side of rows 4-8, and red text `label` and `addr` next to `addr` in rows 7 and 8 respectively.

SerialNo.	Microinstruction	Semantics
1	<code>mloadIR</code>	$ir \leftarrow [pc]$
2	<code>mdecode</code>	populate all the decode registers
3	<code>mswitch</code>	jump to the $\mu pc$ corresponding to the opcode
4	<code>mmove reg1, reg2, &lt; args &gt;</code>	$reg1 \leftarrow reg2$ , send the value of <code>args</code> to the unit that owns <code>reg1</code> , <code>&lt; args &gt;</code> is optional
5	<code>mmovei reg1, imm, &lt; args &gt;</code>	$reg1 \leftarrow imm$ , <code>&lt; args &gt;</code> is optional
6	<code>madd reg1, imm, &lt; args &gt;</code>	$reg1 \leftarrow reg1 + imm$ , <code>&lt; args &gt;</code> is optional
7	<code>mbeq reg1, imm, &lt; label &gt;</code>	if $(reg1 = imm) \mu pc \leftarrow addr(label)$
8	<code>mb &lt; addr &gt;</code>	$\mu pc \leftarrow addr(label)$

Mc Graw Hill Education

So, let us summarize the 8 instructions that we introduced the 8 introductions that we introduced our like. This that we introduce the `mloadIR` micro instruction which takes the contents of the instruction memory address by the `pc` and puts them in `ir` `mdecode` populates all the decode registers `mswitch` will jump to the relevant micro program counter corresponding to the opcode as the simple risk instruction.

Then we define 3 instructions which actually use the arguments in they can have different variance with or without arguments, but let us assume that even if arguments are not there is still part of the same generic variant for the arguments are not their right. So, for example, in a `mmove` micro instruction which takes register 1 and register 2. We said register 1 the contents of register 1 to the contents of register 2 and these are all microregisters, and we send the value of `args` right. So, we send the value of `args` on the bus in specific we send it to the unit that owns `reg 1`. For example, if you send it to `aluResult` it will essentially go to the ALU.

So, since send on the bus theoretically everybody can read, but the unique that owns a certain register is the one which is supposed to read and process it. Similarly, we have the mmovi instruction which moves in the immediate to a register. And so here also args is optional, but it works in exactly the same manner, where args is sent on the bus and the unit that owns reg one and some processing it. We have madd which is re which does the same thing it such reg one as reg one plus m args is again optional, but it does the same thing we have 2 kinds of branches mbeq and mb mbeq is conditional upon the quality that if a given register is equal to an immediate then we said the micro pc to the address of the label well I would not put an address here I would rather put a label. So, that is the mistake.

So, here at whatever is the address corresponding to the label in the micro program right, that will be set to the value of the micro pc and mb is an unconditional branch. So, so that is an unconditional branch. So, in this case we will jump to the label in any case. So, the micro pc is said to the address of the label. So, these are the only 8 instructions that we introduce and we will use these 8 instructions to do all our work.

(Refer Slide Time: 35:20)

**Implementing Instructions in Microcode**

\* The microcode preamble

```

.begin:
  mloadIR
  mdecode
  madd pc, 4
  mswitch
  
```

*pc = pc + 4*

- \* Load the program counter
- \* Decode the instruction
- \* Add 4 to the pc
- \* Switch to the first microinstruction in the microcode sequence of the prog. instruction

*add* *sub*

Mc  
Graw  
Hill  
Education

53

So, every micro you know every instruction in it is micro program will always have some instructions in common at the beginning that is called the preamble right of every micro program which is common. So, this preamble we have made it start at the dot

begin label. So, the assembly format is also the same as simple risk which we introduced in chapter 3.

So, here we start with `mloadIR`, we have to because we need to read the contents of the current pc then we decode. So, this is again common then we call the a madd instruction to increment the pc by 4 bites to go to the next simple risk instruction. So, basically we have  $pc = pc + 4$  over here and which means next time that we read the program counter it will be the next instruction next simple risk instruction, and then we start executing the loaded micro program, then we actually load the pro micro program and we call the `mswitch`. So, so `mswitch` does many things what it does is it switches to the first micro instruction in the microcode sequence of the program instruction of the simple risk instruction. So, it does 2 things one thing is given the simple risk instruction it first in a figures out which micro program with supposed to execute.

Then it also moves the micro pc to the first instruction in that micro program. And for example, `add` will have a separate micro program `subtract` will have a separate micro program. So, whenever we encounter `subtract` and we have decoded it and recall `mswitch` it will move the micro pc to the first micro instruction in the micro program in the part of the micro program that is specific to `subtract` simple risk instruction. So, the preamble remains commons. So, all instructions will have to execute the preamble first which have these 4 micro instructions these are absolutely common after this term micro instructions will be different.

(Refer Slide Time: 37:46)

### 3 Address Format ALU Instruction

*add r1, r2, r3*  
*r1 = r2 + r3*

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

Mc  
Graw  
Hill  
Education

54

So, let us now look at the micro code of different high level instructions; high level meaning simple risk instructions. So, let us first considered 3 address format ALU instructions something of the form  $add\ r1\ r2\ and\ r3$ .

Something of this form where we actually do  $r1\ is\ equal\ to\ r2\ plus\ r3$ . So, what we do is that we first read all the values we transfer them to the ALU and then we perform the ALU operation. So, the first step is we need to read the sources of the ALU operation which in this case in a running example is in our operation, but can be any 3 address format ALU instruction, then we do the ALU operation and we write the result back. So, there are many steps here let us take a look. So, the first step which is first 2 lines is to transfer the first operand to the ALU. So, what we do is that we issue and mmov instruction. The mmov instruction accesses the rs 1 register the first source and then it transfers it is containing to the regSrc.

So, recall that the register files the regSrc instruction encapsulates it and it contains idea of the registers that needs to be accessed. So, we take the contents of rs 1 and put it in regSrc, and further what we do is that we send the arguments read on the bus. So, which means that 2 things will happen what will happen is that the contains of rs 1 will go to the register file will be saved in regSrc, simultaneously the read operation will be initiated in the register file and the register file will take the values from the regSrc field.

So, think of it this way on the shared bus the register file is connected, 2 things are coming together one is that read operation is coming from the bus also what is coming is that the contents of rs 1. So, the contents of rs 1 will actually be the register r 2. So, the register r 2 you know that idea is coming it is going into the regSrc microregister and simultaneously it is going into the register file right and so basically we are doing a register file read access also simultaneously.

So, we are doing 2 things. So, you know may be this diagram as small. So, let me draw it once again this is very important appreciate that we are actually doing 2 things over here. So, let me draw the shared bus once again. So, what we are doing is that and also here we have the rs 1 register. So, what we are doing is that we are executing an assembly instruction. So, which in hardware is telling the rs 1 microregister to send its value on the shared bus.

And once it is something is sent on the shared bus everybody will be able to pick it up we are also telling that let us send read command on the shared bus as a part of the arguments. So, then the register file will see that the read command is meant for itself. So, it will activate itself and it will see that you know read is being sent along with that the contents of rs 1 are being sent.

So, it will do 2 things we are also asking the register file to move the contents of rs 1 to regSrc. So, it will do that the contents of rs 1 will be moved to regSrc, in addition since we have sent the read command the register file will take the contents of rs 1 and along with writing it to regSrc in parallel it will start performing the register file read access say it will do 2 things write to regSrc and since arguments have been set it will use the contents of rs 1 light, which it will treat as the source contents the which will treat as the idea of the source register to access the register file. Then it will store the final result the computed result in the macro register regVal by the end of the cycle.

Subsequently what we will do is that we will move this to register A which is a part of the ALU system right ALU has 2 input registers A and B.

So, whatever is there in regVal in the next cycle we will transfer the contents of regVal 2 register A again why are the shared bus, but since there are no arguments nothing needs to be done. So, let me erase the ink on this slide and let me move to the next part.

(Refer Slide Time: 43:51)

### 3 Address Format ALU Instruction

```
/* 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>

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

The diagram illustrates the hardware components and data flow for the ALU instruction. A central 'Shared Bus' is shown with arrows indicating bidirectional communication. On the left, the 'ALU' is connected to the bus. On the right, the 'Register File' (RF) is connected to the bus. The bus is also connected to the 'mmov regSrc, rs2, <read>' instruction, which reads from the register file. The bus is also connected to the 'mmov B, regVal, <aluop>' instruction, which writes to the register file. The bus is also connected to the 'mmov B, immx, <aluop>' instruction, which writes to the register file. The bus is also connected to the 'mmov regSrc, rd' instruction, which reads from the register file. The bus is also connected to the 'mmov regData, aluResult, <write>' instruction, which writes to the register file. The bus is also connected to the 'mb .begin' instruction, which writes to the register file. The bus is also connected to the 'mmov regSrc, rs1, <read>' instruction, which reads from the register file. The bus is also connected to the 'mmov A, regVal' instruction, which writes to the register file. The bus is also connected to the 'mbeq I, 1, .imm' instruction, which writes to the register file. The bus is also connected to the ':imm:' instruction, which writes to the register file. The bus is also connected to the 'mmov regSrc, rd' instruction, which reads from the register file. The bus is also connected to the 'mmov regData, aluResult, <write>' instruction, which writes to the register file. The bus is also connected to the 'mb .begin' instruction, which writes to the register file.

Mc Graw Hill Education

54

So, here what we do is we check the value of the immediate register. So, we compare the contents of register i with the value 1. If i is equal to 1 it means of the immediate is said to true. So, we will jump to dot imm which is over here what we do is that we transfer the contents of immx which is the extended immediate to the register B. So, recall that the A leave is also connected to the shared bus why are the registers A and B in the args are also coming in and this is the ALU and we also we can in optionally send this data here if it is a simultaneous thing which is exactly what we are doing here.

So, we are sending immx on the bus immx is being written to microregister B in addition we are sending the ALU op in a ALU op argument which is basically what is operation we want to do on the ALU which can be add subtract multiply all other in a compare all other ALU operations these ALU operations are being send to the ALU. So, when the ALU sees that a command via the args field is being send to it activate itself and it takes a look at what other data is being sent. So, the value of immx is being send and this is being sent to register B. So, along with writing it to microregister B it will also go to the ALU and since A was already populated over here both the operands are there in A and B. So, the ALU will start doing the ALU operation.

Let us take a look at the other branch of this statement if, if the immediate field was 0 which means the second source operand is a register right is a second operand with the register then we need read the contents of rs 2 which is sees exactly this statement

repeated once again that we read the contents of rs 2 transfer them to regSrc, regSrc is the microregister that encapsulates register file simultaneously we send the read argument on the shared bus. So, along with writing the value of rs 2 to regSrc, it will also read the register file with whatever value is being sent that output will be written to the regVal microregister.

So, again statement similar to this will be executed here. So, since the value of A is already there and it has been set in advance and now you know some ALU operation add subtract multiply whatever is coming via the bus, and we are writing some value is being written to microregister B, we can write it to macro register B and also in parallel send to the ALU and the ALU will then process the contents of A and B and the ALU operation and write it is result. So, after this we move to the next micro instruction and so we do not want to enter this part. So, we will directly make a jump and unconditional jump to the rw label. So, if the rw label is here. So, will execute this instruction what we will now do, is that we will start the procedure to write the aluResult to the register file.

So, let me do a partial erase. So, in the register file we have the regSrc microregister which contains the source, which contains the idea of the register it needs to be excised and we are the regData macro register, which contains the data that needs to be written and then again of course, we have the shared bus. So, what we do is that we move the contents of the rd register which contains the idea of the resonance register to regSrc in a first cycles of the we do not access the register file that because there are no arguments. So, there is no command, but we populate the regSrc microregister with rd write the idea of the destination register.

Since the result of the ALU operation is there in the aluResult microregister, we transfer this to regData via the shared bus of course, So, aluResult which can which has the result of the last ALU operation, it zeta is transfer to regData because this is what needs to be written. And since the id is already there we can simultaneously send a right signal via the arg args argument to write to the register file. So, in this manner what we do is that we first read data from the register file in these statements or we read the immediate I am sorry with this statement and this statement or we read the immediate over here, we perform the ALU operation and then we write the result of the ALU operation back to the register file.

Once we are done we make an unconditional jump to the begin label and recall that the begin label is the beginning of the preamble. Where there is first fix set of instructions that we have to execute and then we move to the particular microcode microprogram I am sorry of each and every simple risk construction.

(Refer Slide Time: 50:08)

**3 Address Format ALU Instruction**

```

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

/* check the value of the immediate register */
mbeg 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>

/* write the ALU result to the register file*/
.rw:
mmov regSrc, rd
mmov regData, aluResult, <write>
mb .begin

```

*regSrc ← rs2*  
*read/write*  
*RF*

54

So, now the important point of course, see large part of this is very straight forward of just moving things around, but the important point is array instruction of this type. So, recall that 2 things are being done here. First we are setting the contents of regSrc to the contents of rs 2 that is one thing in parallel I remind you in parallel simultaneously concurrently, we are also accessing the register file. Why are we doing that because the command is being send why are bus, why are the args field of the bus to the register file to either read or write in this case it is read does not matter it can be read or write since a command is being sent we will activate the register file and it since it is a read operation.

We will read the data from the regSrc microregister. So, we allow a concurrent writing of this microregister now also reading his values we will need to have a very small powering circuit the that is very simple detail I am ignoring that, but pretty much the regSrc microregister needs to be accessed we need to get it is current contents, if it current contents are being written to have been modified that is if whatever we are writing you can sort of bypass the value, that is and then we can start the case. So, in this case we say one cycle. So, this is an interesting need elegant efficient optimization to



make, we do the same thing for an ALU operation as well where we also right the value of regVal to microregister B and also simultaneously initiated in ALU operation.

So, this can be done and you know with the small forwarding path this can be done. So, this is per say and not a very technically challenging operation.

(Refer Slide Time: 52:14)

The slide is titled "The mov Instruction" and contains assembly code for the `mov` instruction. The code is annotated with red handwritten notes and arrows. A diagram of the register file is also shown.

```
mov instruction
/* check the value of the immediate register */
mbeq i, 1, .imm
/* second operand is a register */
mmov regSrc, rs2, <read>
mmov regData, regVal
mb .rw
/* second operand is an immediate */
.imm:
mmov regData, immx
/* write to the register file */
.rw:
mmov regSrc, rd, <write>
/* jump to the beginning */
mb .begin
```

Handwritten annotations include:

- Red arrows pointing from `mbeq i, 1, .imm` to the `.imm:` label.
- Red arrows pointing from `mmov regSrc, rs2, <read>` to `regSrc` and `rs2`.
- Red arrows pointing from `mmov regData, regVal` to `regData` and `regVal`.
- Red arrows pointing from `mmov regData, immx` to `regData` and `immx`.
- Red arrows pointing from `mmov regSrc, rd, <write>` to `regSrc`, `rd`, and `<write>`.
- Red arrows pointing from `mb .begin` back to the start of the code block.

The diagram shows a register file (RF) with two registers, `r1` and `r2`. `r1` is labeled "write" and `r2` is labeled "read".

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

So, now that we have seen a difficult set of instructions namely 3 address format ALU instructions. Let us take a look at something similar yet significantly simpler namely the `mov` instruction. So, this is easy we first check the value. So, since I am also discussing the format of most of the instructions I will go slightly faster. So, we check the value of the immediate register how do we check the compare `i` with 1. So, we read the value of the `i` microregister which has been populated in the `mdecode` step.

We compare it with 1 if the comparison is successful which means `i` contains 1 we directly jump to `dot in`. So, what we do is that the contains of `immx` right extended immediate or written to `regData`. So, `regData` recall is the register which contains the data that needs to be written to the register file. Otherwise the second operand is a register right because immediate is 0. So, what we do is that we read the contents of `rs 2` and transfer it to `regSrc` and simultaneously we also send the read command on the bus. So, it initiates the registers file read and the results of the read or store in the `regVal` micro microregister right. So, the register file this is the register file we have `regSrc`

which is the idea of what needs to be read, and we have regData which is the contents of what need to be written and then we have the output of the register file as regVal.

So, what we do is that we send the contents of rs t. So, we read the read one of the registers right the second register second operand, then what we do is once we read it we again need to use it to write it to the resonation. So, we transfer the contents of regVal to regData by this move instruction. Then we do an unconditional jumped to rw which brings us over here. So, since the correct id is loaded in regSrc not yet not I am sorry. So, then what we need to do is we need to load the idea of the destination register in the regSrc microregister. So, what we do is we read the microregister rd which contains the exactly this information, and this is transfer to regSrc and since in advance we have already written the right data that needs to be written via this statement over here and this statement over here.

Either the contents either the immediate or the contents of a register have been written to regData, we can in parallel initiate register file right by sending the right command which is exactly what is being done, and this what we have done is that if let say we want to set r1 to r 2 we essentially read the contents of r 2 and transfer it to the destination register r1 which we have just done we have either read a register or an immediate and we have written it to the destination register once we are done we jump to the beginning at the preamble by the micro instruction mb dot begin.

(Refer Slide Time: 56:03)

### The not Instruction

*not*  
~~move~~ instruction

```
/* check the value of the immediate register */
mbeq I, 1, .imm
/* second operand is a register */
mmov regSrc, rs2, <read>
mmov B, regVal, <not> /* ALU operation */
mb .rw
/* second operand is an immediate */
.imm:
mmov B, immx, <not> /* ALU operation */
/* write to the register file*/
.rw:
mmov regData, aluResult
mmov regSrc, rd, <write>
/* jump to the beginning */
mb .begin
```

Mc  
Graw  
Hill  
Education

56

Now let us take a look at on not instruction a little bit a problem here they should be not. So, here we do exactly the same thing, but we did need to do a little bit more. So, this is essentially 2 address format instruction. So, we do a little bit more, but recall the, but try to understand that all the programs are written more or less in a similar fashion will little bit of difference here and there.

So, what we do is we check the value of the immediate register. So, we again compare it with 1, if the second operand is 1 a register we do something otherwise you do something else. So, we have always been looking at the first case you know the case when the operand is an immediate. So, let us take a look at it first if it is some immediate what we do is that we access the immx microregister which contains the immediate we move it to the ALU microregister B.

Why because recall that the ALU is encapsulated with 2 microregisters A and B and we have always in an even back in capture 3 we have use B as the source of data for the not instruction. So, we will do the same over here we will populate the B microregister it will send it is containing the ALU and since this is the single operand ALU instruction we can in parallel initiate the ALU operation as well by sending the command not write the not ALU instruction on the bus. So, the ALU in 1 cycle will be ready with it is result; however, if the second operand is not a register is not an immediate it is a register we need to do something else which is that we need to access the rs to microregister which contains the idea of the source operand, we need to this is the register id. So, this needs to be transfer to red source as we have been doing.

And in parallel we can issue a register file read. So, in one cycle the data will be ready in the regVal register. So, this we need to do exactly the same as we did here in this line that regVal will send it to the ALU register B and in parallel also issue an ALU operation to compute the not of the value. Once done once we are ready with the aluResult we do an unconditional jump to RW. So, irrespective of how we get here. So, let me in our slide is getting cluttered up; so little bit of cleaning is due. So, irrespective of how we get here what we know at this point is. So, the ALU has computed it is value and the value is present as of now in the aluResult in the aluResult microregister and the value can come from 2 sources it can either come from immx which can tends the immediate or it can come from the register file.

So, if the id of the source register is in rs 2. So, in both of these cases we have read we have transferred the data to the ALU with the command specifying the do compute the not operation. So, once done what is left we need to right the result to the register file. So, we transfer the contents of aluResult to the microregister that contains you know what exactly needs to be written to the register file which is regData. So, the regData micro reg is populated here. And we read the rd microregister that contains the id of the destination register this is send to regSrc. So, there we have a register file and the register file has in reason f it is an f. So, it has red source for the id of the register and regData of our, what is that that needs to be written.

So, regData came from the output of the ALU which is the aluResult, and regSrc will come from the rd register which contains the id of the destination register. So, since regData is already populated we can in parallel issue a write in the register file and the write will happen in one cycle, and then we are done. So, we jump to the beginning of the preamble. So, we have done instructions of the form move or not and also 3 address format ALU instruction.

(Refer Slide Time: 61:12)

The slide is titled "The cmp Instruction". It contains the following assembly code with handwritten annotations:

```

cmp instruction
/* transfer rs1 to register A */
mov regSrc, rs1, <read>
mov 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, <cmp> /* ALU operation */
mb .begin

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

```

Handwritten annotations include:

- cmp rs1, rs2, immx* written in red above the code.
- Red circles around *rs1*, *regVal*, *rs2*, *regVal*, *cmp*, and *flags*.
- A diagram on the right shows a box labeled "ALU" with two inputs from above and one output to a box labeled "flags".
- A red arrow points from the *flags* box back to the *mb .begin* instruction.

The slide also features the "Mc Graw Hill Education" logo in the bottom left corner and the number "57" in the bottom right corner.

So, let see what is left they compare instruction. So, the compare instruction is also simple. So, now, that we have got in a hang of micro programs my speed of going through these programs is slightly increase just in case you are not able to understand me

my advice would be that please go back you know one slide 2 slide there slides and then try to understand that part.

Also read the relevant sections in my book and after that you will be able to understand these things. So, they are very simple, but you just need to get a hang of it. So, the compare instruction same thing, compare instruction recall that it has to we do not have a destination it has 2 sources, the first source is always the compare instruction is always at the form it has one source in rs 1 the second source can either be rs 2 or it can be an immediate right immx. So, let us first read rs 1. So, we transfer the contents of rs 1 to regSrc right. So, the id of what needs to be read that is into regSrc simultaneously read command is issued. So, in one cycle the result is ready result moving the contents of rs 1 from the register file is ready in regVal which is transferred to the first register microregister in ALU which is A and B.

So, recall that for the ALU there is 2 microregisters A and B and the ALU compute some operation on the contents of A and B. So, for a we have loaded the first argument the contents of rs 1. Same thing once again we check the value of the immediate if it is one we do something if we 0 we do something else. So, we always had been explaining the one case first. So, let us try to strict with the same tradition if it is an immediate well is very simple we move the contents of immx why are the shared bus of course, 2 b and since A is already p populated right over here we send the compare command on the bus right why are the args mechanism we send the compare command so the ALU will populate B in parallel also do the comparison in one cycle will be done.

However, if the second source is the register no problem let us read rs 2 let us you know to read it we save rs 2 in regSrc and also issue the read command. In one cycle the result is in regVal which is send to B write B again. So, B can either gets it is value from the immediate from immex or from the register file does not matter and we simultaneously issue the compare command to the ALU and so we are ready with. So, in your doing a comparison the aluResult will contain nothing. What will happen is as a results of the comparison will prêt will populate the flags register. So, we can read flags dot E and flags dot GT at a later point of time to figure out what was the result of the comparison.

So, the compare instruction will not write to any register. So, it does not have any register side effect per say it will just modify the flags. So now, that we have so. Now

basically what we do is in this path once we have done the comparison we can actually jump at the preamble, in the other case once we have done the comparison where can also jump at to the preamble and we thus exist.

(Refer Slide Time: 65:12)

The nop Instruction

```
* mb .begin
```

Mc  
Graw  
Hill  
Education

58

Let us continue our journey. So, nop instruction well this cannot be simpler you will be really having it to the see the code, we do nothing at all we just jump to the beginning of the preamble and we have all said to load the next simple risk instruction.

(Refer Slide Time: 65:32)

The ld Instruction

```
id instruction
/* transfer rsl to register A */
mmov regSrc, rsl, <read>
mmov A, regVal

/* calculate the effective address */
mmov B, immx, <add> /* ALU operation */

/* perform the load */
mmov mar, aluResult, <load>

/* write the loaded value to the register file */
mmov regData, ldResult
mmov regSrc, rd <write>

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

SB  
ldResult  
mmov  
mmov  
mmov  
DM

Mc  
Graw  
Hill  
Education

59

We just jump to the beginning mb dot begin. Now let us look at the load store instructions right the memory instructions. So, there also very easy everything is very easy in out set up right nothing is hard at all, in if you thing is hard is just in the mind. So, the load instruction works I am sorry instead of I read it should be l. So, let me just note down the slide numbers such that I can fix it. So, the load instruction works like this that the format of the load instruction is that, we actually write to a destination register and we have an immediate which is contains the offset and rs 1 contains the rs 1 is the base register. So, this is the format of the load instruction.

So, what we do is we read rs 1 first. So, we read it by transferring the contents of rs 1 to regSrc, and by transferring the read command on the bus. So, then the output of the register file will have the contents of you know some register whose id was in rs 1, this is reg val. So, what we do is we transfer this to the ALU and so basically the ALU which has to operands A and B; one of the operands is the contents of rs 1.

Subsequently what we do is we calculate the effective address which is basically the contents of the rs 1 registered plus immx right, well not the rs 1 registered, but essentially the rs 1 contents of id of registers; so the contents of that register plus immx. So, what we do is we move immx to B. So, pretty much we move immx to B and in parallel we issue an ALU add operation to compute the memory address. So, we are adding a which is the base address and B which is the memory off set. So, we are adding both of them via the ALU.

So, the result of the addition will be saved in the microregister aluResult right. So, then of the, what we will now to is that using the shared bus, we will transfer the address aluResult to the memory unit. So, recall that in the memory unit. So, may be let me just in erase and redraw let me erase of all of it. So, recall that in the memory unit we have the data memory. We have the mar for the memory address register and the mdr which is the memory data microregister, but mdr in this case is not required. So, what we actually do is why are the shared bus that me call it SB we transfer the aluResult which contains the address of the address of the memory location, this we transfer to the mar register and in parallel we initiate a memory load.

So, we send the load command. So, we send the load command and we write to the mar register. And also this data is send to the data memory and in parallel we send the load

command as well. So, in one cycle we expect that the data memory is accessed and the result of the load is available in the output register of this microregister of this particular stage which is the ldResult in a microregister. So, ldResult contains the output of the load well. So, what is left, what is left is that we need to write the contents of ldResult to the register file. How we will do that well that is very easy we have done at many times now we transfer the contents using an mmov instruction we transfer the contents to the regData microregister right. Because this contains the data that needs to be written in subsequently in the next micro instruction, we read the idea of the destination register rd write. So, we read it is id and we transfer that your regSrc, because regSrc is supposed to contain the idea of the register and in parallel we issue the write command.

So, the write command will essentially write the ldResult to the destination register, write in the register file. So, since we have discussed its mechanism few times before I am not going in to details, but pretty much to summarize the load operation. We first read the contents of the base register write it to the ALU. We read the immediate which is the memory offset we write that again to another ALU register and also simultaneous the initiate and addition. The addition results are the address is the memory address this is sent to the memory unit and in parallel, we initiate a memory load. After that what we do is once the ldResult is available we write it to the register file and the idea of the register that we will write it to comes from the rd field.

So, once the load is done we can jump at to the beginning of the preamble by doing an unconditional jump to the dot begin label.



(Refer Slide Time: 71:29)

### The st Instruction

```
st instruction
/* transfer rsl to register A */
mmov regSrc, rsl, <read>
mmov A, regVal

/* calculate the effective address */
mmov B, immx, <add> /* ALU operation */

/* perform the store */
mmov @mar, aluResult
mmov regSrc, rd, <read>
mmov mdr, regVal, <store>

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

The diagram illustrates the microcode for the 'st' instruction. It shows the flow of data between various registers and memory components. Key components include the Shared Bus (SB), Memory Address Register (MAR), Memory Data Register (MDR), and Data Memory (DM). Red annotations highlight the 'store' operation and the flow of data from registers to memory.

Mc  
Graw  
Hill  
Education

60

Well we did load. So, symmetry: such that we have to do store. So, stores have the same format technically, but recall that you know store a little bit of an exception was made. In a sense that store does not have a destination it rather has to sources and that is fine one of them is the address source on in the other is the data source. So, this is also a register which contains the data that needs to be stored in the memory address. So, we have discussed the store exceptions, let me say how this is the implemented in micro code.

So, what we do is that we will we do the same 2 2 lines are common. We transfer the contents of rs 1 to ref source and we initiate a register read and the contents of the base register go to the ALU registered A. We subsequently read the value of the immx, immx is the offset the immediate this is send to the ALU via it is microregister B. We then initiate and add operation in parallel. So, the aluResult contains the memory address, and the memory address is then sent to the mar register on the memory address register. So, recall that the memory access stage with the data memory DM for data memory is organized like this that we have a mar and mdr microregisters. And of course, some arguments at come from the shared bus SB for shared bus.

So, in the mar register we save the result of the ALU operation with gives us the memory address and for the regSrc. So, regSrc is the idea of the register that we need either need to read to write to. So, what we do is we populate this how do we populate it well. So, what we do is that we read the idea of the rd register. So, in this case the rd does not

contain the destination rather it contains the data that needs to be is told. So, any case we need to read it. So, we read it by transferring it is contents to regSrc and initiating the read command after that the value is there in reg val. So, this is transfer to the memory data register mdr. So, basically the value comes to the mdr register right mdr.

So, this is the value that needs to be stored in the data memory. So, in parallel you know we in the previous case we are a load command that we are issuing in this case we issue us store command store to store to the data memory DM. So, this what will I do since the memory address has been pre populated in this line. So, let me we have do a little bit of erasing. Since the memory address has already been populated in advance by the contents of in our ALU operation, now what we are doing is we are reading the next operand which is rd reading it is contents from the register file and transferring it to the memory data register which contains the data that needs to be written to the data memory in a store operation. And since the rest of the arguments are all there we can in parallel initiate a store operation via the rth field.

And that will store the data to the data memory well then we are run. So, we just have to then you know migrate back jump back to the beginning of the preamble for the next simple risk instruction.