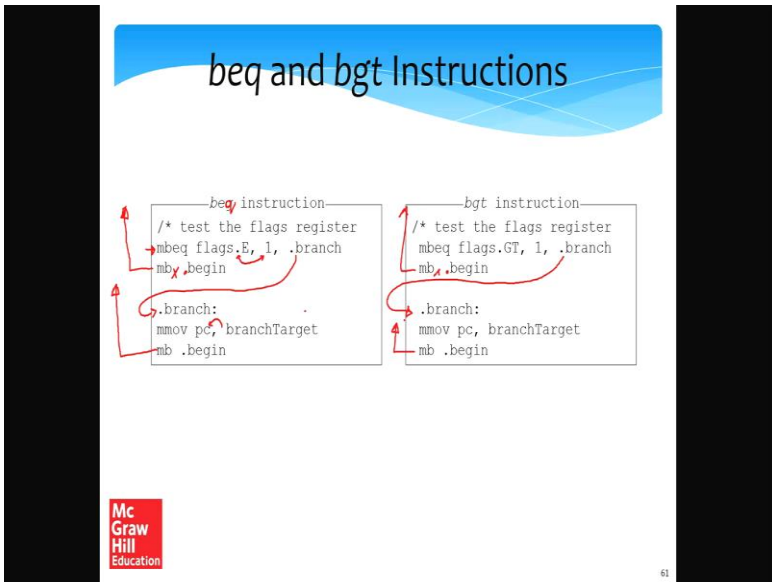
Computer Architecture

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

Lecture - 25
Processor Design Part-III

(Refer Slide Time: 00:25)



beq and bgt Instructions

beq instruction

```
/* test the flags register
mbeq flags.E, 1, .branch
mb .begin

.branch:
mmov pc, branchTarget
mb .begin
```

bgt instruction

```
/* test the flags register
mbeq flags.GT, 1, .branch
mb .begin

.branch:
mmov pc, branchTarget
mb .begin
```

61

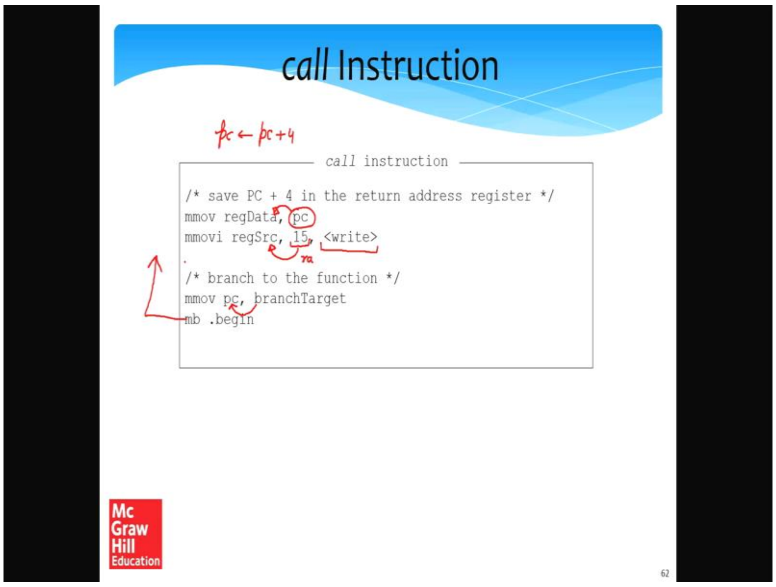
Let us now discuss the implementation the beq and bgt instructions. So, recall the beq is branch of equal. So, equal means the equal flags dot E flag bit set. Means that the last comparison resulted in an equality, and flags dot GT is used for the bgt instruction. Means, the last comparison resulted in one number; the first operand being greater than the second operand. So, let us look at beq first. Sorry this should be q all right. So, first thing that we do is that we test the flags register.

So, we invoke the m beq micro instruction to test flags dot E. So, recall that flags dot E is also a micro register; that is exposed on the shared bus. So, we compare flags dot E with one. If there is equality then we jump to the dot branch label otherwise. So, otherwise pretty much you know if there is no equality, this instruction is as good as no (Refer Time: 01:42). So, then again proceed to the next instruction. So, we jump to mb dot begin which means, sorry this should again be here mb dot begin, which means that we move to the beginning of the preamble.

So, if let say we are branching, then what we need to do, is that we have already computed the branch target. So, recall that the branch target, once computed in the decode stage. So, the branch target was saved in a separate micro register. For non branch instructions it contains junk, but for branch instructions it contains a valid branch target. So, we move the branch target to the pc register, and then again we jumped to the next instructions. So, beginning of the preamble, but this time when the preamble execute, it will see an updated value of the pc which is the target of the branch, not old pc plus 4. So, it will start execution from that point.

The bgt instruction is exactly the implemented in exactly the same way. It is just that instead of comparing flags dot E with one. We compare flags dot GT with one, if there is equality we move to dot the dot branch label; otherwise we move to the preamble, and then we move the branch target to the pc, and we start executing preamble for the next instruction at the branch target one again.

(Refer Slide Time: 03:17)



call Instruction

$pc \leftarrow pc+4$

call instruction

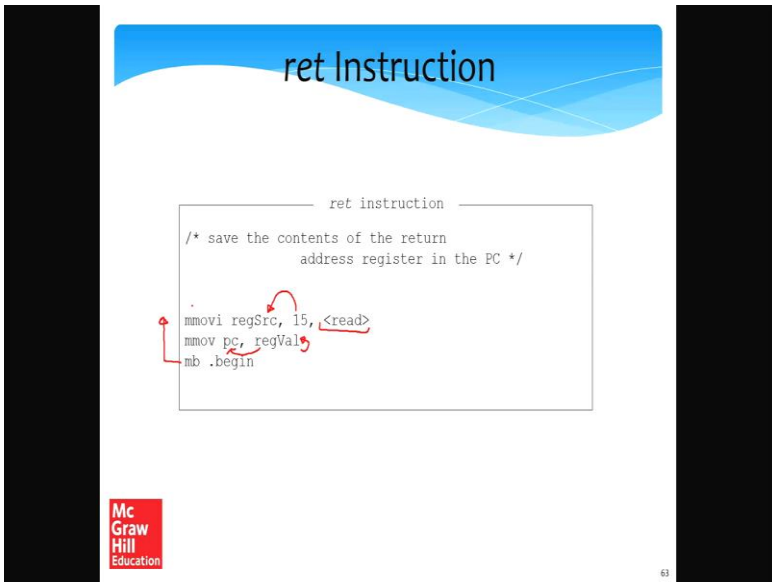
```
/* save PC + 4 in the return address register */
mmov regData, pc
mmovi regSrc, 15, <write>

/* branch to the function */
mmov pc, branchTarget
mb .begin
```

Let us now take a look the call instruction. So, the call instruction is very similar to the branch instruction, is just that little bit of more works needs to be done. So, what we need to do is that we need to save the value of the pc, after the call instruction the next pc into the written address register. So, if you recall in the preamble. So, in fact, in the third statement of preamble, we are actually incremented the pc by 4. So, we are set the current pc equal to the old pc plus 4. So, that we are already done. So, what is there in the current pc is actually the old pc plus 4. So, we do not have that incremented once again. We can directly write it to the written address register. So, what we do is, that we transfer the contents to the regData micro register. Then this is what is supposed to be written to the register file, and then we transfer 15 to regsource. So, why 15, because 15 is the id of the written address register, is the id of r a, and simultaneously we sent the right command on the shared pass.

So, basically what we are doing, is that we are writing the value of pc, the value in the pc register, which is the current pc plus 4, which is essentially the address of the next instruction, to the register file, and which register the register file register 15 r a. Once we have done that we can jump to the branch target. So, we load the branch target in the pc which is equivalent to a jump, and we move to the beginning of the preamble such that we can execute the next instruction.

(Refer Slide Time: 05:08)



ret Instruction

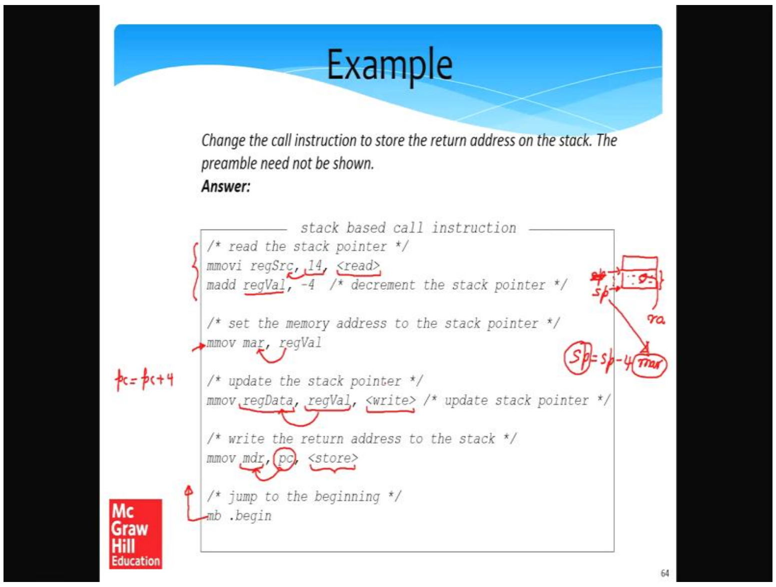
ret instruction

```
/* save the contents of the return
              address register in the PC */

mmovi regSrc, 15, <read>
mmov pc, regVal
mb .begin
```

Let us next look at the ret instructions which is nice and simple. So, what this does is that we read the return address register number 15, and save it is content in the program counter pc. So, what we do is that we load 15 into regsource with the mmovi instruction then mmovi immediate, and we simultaneously send the read command on the bus, so we read the register. The data will be available in the microreg micro register regVal; we should essentially contain the return address, this we move to the pc. So, once we have done that, we can again jump to the beginning of the preamble, and execute the instruction at the return address.

(Refer Slide Time: 05:59)



Example

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

Answer:

```
stack based call instruction
/* read the stack pointer */
mmovi regSrc, 14, <read>
madd regVal, -4  /* decrement the stack pointer */

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

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

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

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

Let us now consider an example. So, let us change the call instructions, to store the address on the stack, instead of the in the r a return address register. So, we will just show the microcode, the preamble will not be shown, because it is common to all the instructions. So, stack based call instructions is like this, that the first thing that we need to do is, that we need to read the value of the current stack pointer, then we need to decrement the stack pointer and store the return address there, and we need to return.

So, let us do, let us read the stack pointer first. So, the first thing that we need to do, is that we move the id of the stack register, the stack point register 14 into micro register regsource, and simultaneously we do a read command. So, we are basically reading the value of the stack pointer from the s p register. Once we have done that, the value of the stack point will be in the regVal register, which is the output of the read instruction of the read micro instruction.

So, now what we do is that we will decrement the value of the regVal by minus 4, which essentially means that we are creating some space on the stacks. So, if say let say this is the last element on the stack; we are decrementing the stack pointer by 4 bytes, which is the size of an integer in the 32 bit system. And we are creating the some new empty space where something can be stored. So, we are essentially decrementing the stack pointer here. So, we read it first, then we use the m add instruction to add minus 4 to it. So, we decrement the stack pointer. And once we decrement the stack pointer, this becomes the new value of the stack pointer.

So, what we simultaneously do is that since we need to store something over here. We send this to them mar memory address register, in a mar micro register. So, we send it there using simple mmov call. So, we send the updated value of the stack pointer which is the old sp minus 4, to the mar micro register, using an mmov instruction. So, this is to be required. So, I will use store the return address.

Now, let us come back now that the stack pointer has been updated, it needs to be return back to the register file right. So, what we will do is that we will write it back. So, since we will write it back which register are we writing it back, we are writing it back to whatever is stored in regsource. Since 14 the id of the s p register has been stored in regsource, and we are subsequently not changed regsource. So, the value 14 will remain, the stack pointer will remain. regVal has been updated with this mar instruction, where we have essentially computed the new stack pointer, which is old s p minus 4.

So, this value will be written to return back. So, the way we will do it is that we write it to regData which contains the data that needs to be written, and simultaneously we will issue the write command. So, this will update the value of the stack pointer inside the register file, and the new stack pointer will be the old stack pointer minus 4. So, the register file will thus contain the updated value.

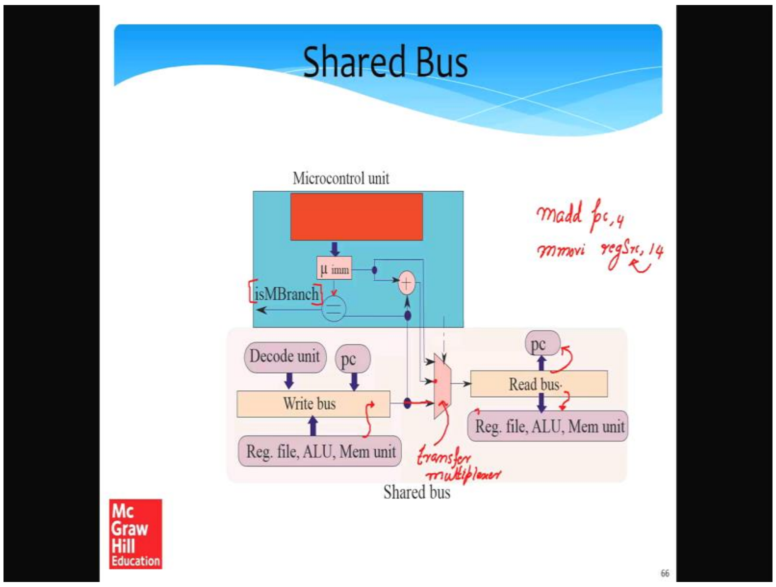
Once that is done; so we need to write the return address, to the you know to the new location on the stack that we have just created, and the return address will be there in the pc register, the reason being that if you recall the third or the fourth instruction in the preamble was that we are incrementing the pc by 4 right. So, pc is equal to pc plus 4. We already had this part in the preamble, so may be quickly go back to the preamble and show you. So, this is where we are m add pc 4.

So, back, all right. So, we have already incremented this by four. So, this contains a return address. So, we move this to the m d r micro register, and simultaneously we issue the store command on the bus. So, this ensures that we store it to this location on the stack. So, let us see how. So, the mar register which contains the address, already contains the location of the new stack pointer, this has been done here. So, the address is correct, and now we transfer the data, that what is the data. Well the data is the value of the return address which is stored in the current pc register. This is transferred to the m d r micro register, and simultaneously we issue the store command. So, as a result in a new stack location that has been created the return address is stored over there. Now that we are done we finish the execution, and we jump to the beginning of the preamble by calling the micro instructions mb dot begin.

So, this instruction is. I am sorry, this example is indicative; it is representative of the whole host of optimizations and complex instructions that we can add, with the micro programming mechanism, and we can also change the behavior of current instructions. for example, let say we take the call instruction, we change this behavior to actually write the return address on the stack right, instead of in the r a register, and similarly we will have to modify the ret instruction as well, but it is possible to do all of that, and it is possible to all of that fairly easily.

So, the important point that we are making here is by exposing details of the internal of the processor. We can implement new instructions, make existing instructions work differently, and this can help us add a lot of additional extremely sophisticated mechanisms, to word processors. All right, now that we have seen most of micro programming. Let us quickly take a look at the micro control unit, or the unit that runs these micro instructions.

(Refer Slide Time: 12:36)



So, this is the shared bus we have until we will be looking at it, as one set of parallel wires right; that is the only way that we have been looking at this. We are not been looking at it you know very differently, but now we will sort of create a more complicated structure, and we will see how it works. So, the way that we design the shared bus is like, this that we actually have two buses write bus. So, write bus is the set of copper wires, where all the registers, sorry the micro registers that can write to it or connected. So, all the decode registers like I r and r s 1 r s 2 r d, all of those decode registers the pc register, they are all connected to the write bus.

Similarly, all the register file ALU and MEM unit micro registers are connected to write bus. So, all of them dump their values to the write bus. Of course, all of them are not activated at the same time. So, we have separate activation signals for one. So, only one of these registers is activated. So, an activated state means, that it can write it is value to the write bus, and the rest of them are quite. So, we will need separate control signals for them I am not shown that the details are in the book, subsequently what we do. So, this is one of the inputs from the write bus which can go to this multiplexer. We call this multiplexer the transfer multiplexer in the book, mainly because it transfers you know data from a writer to reader; that is all you know, we do not use any other sophisticated terminology.

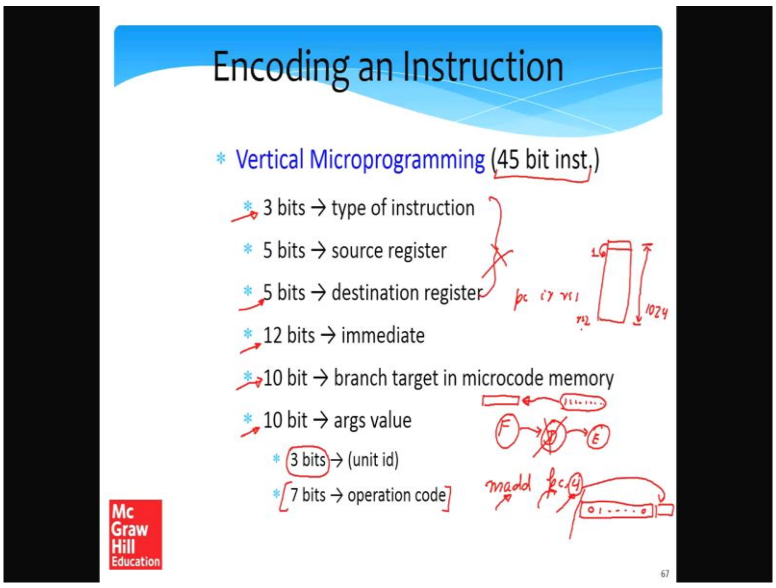
In the addition we have a micro control unit. So, the micro control unit peruses the immediate (Refer time: 14:27) just in case that is a micro immediate, that we are using like a m arrow something or moi. So, in this case, the micro immediate is added to whatever register value is being sent on the write bus, and the added value is the second input, like an m add instruction the added, the result of the addition of the micro immediate with the micro register value, this will be sent to, at the second input of the transfer multiplexer.

So, let me give an example, let us assume that we have a microinstruction of m add pc and 4, so essentially pc will dump it is value in the write bus, and the micro immediate is four. So, both of them will get added, and that will be, the result of the addition of the second input. The immediate it iself can be an input to this transfer multiplexer, and this will be in the case of mmovi which we have actually seen. We have seen where it. This is where we saw right mmovi regsource 14.

So, what mmovi is doing is. So, let us see. So, in this case we take the micro register, and we move a constant to it, and this micro immediate can be one of the value which is being, you know one of the values which is being transferred, from a writer to a reader. So, that is the reason it is also one of the inputs to the transfer multiplexer, which is controlled by a set of control signals, which we shall discuss later. In addition for the, for finding out, whether you know the microinstructions, whether the branch instruction mb q, whether there is an equality or not between the immediate and let say the value in a register. We also have an equality circuit here, which compares the value that, you know compare the value of the immediate, and also the value of one of the registers. So, it compares both. And then what it does is that it generates an equality signal. If there is equality, then ism branch signal is true; otherwise it is false.

After this, so, these are all you know micro registers that are writing to the write bus, on the read bus. The read bus will essentially transfer values to the micro register. So, the read bus will transfer values to the pc and that. So, in the sense the pc can be updated. And how will you get updated; by you know any one of these mechanisms that can be used to update the pc. So, that will read it is new value from the read bus, and also you know there is a micro registers of the type m a r m d r right regVal and so on. So, all the other micro register associated with the register file ALU and MEM unit, they can be updated via values coming from the read bus. So, they will read it from the read bus and update themselves. So, that is the way we will build our circuit.

(Refer Slide Time: 17:54)



Encoding an Instruction

- Vertical Microprogramming (45 bit inst.)
  - 3 bits → type of instruction
  - 5 bits → source register
  - 5 bits → destination register
  - 12 bits → immediate
  - 10 bit → branch target in microcode memory
  - 10 bit → args value
    - 3 bits → (unit id)
    - 7 bits → operation code

So, there are two kinds of, there are two kinds of ways, there are two types of microprogramming; one is called vertical microprogramming, which is, well which has some advantages, and the other is horizontal microprogramming. So, let us consider a micro instruction, how it is encoded. So, this is one particular way of encoding the microinstruction. So, in this case we use 3 bits for the type of the microinstructions. So, we have 8 microinstructions we need 3 bits, 5 bits for the source register, actually if you count we have lot of source registers right. So, basically 5 bits for the source register, 5 bits for the destination register, 12 bits for the micro immediate, 10 bits for the branch target in the micro code memory.

Because we are assuming the memory is not very large, and also from you know. So, we are also assuming the memory is organized in this fashion that in each entry there is 1 microinstruction. So, we do not have to go from one entry to the other, we do not have to add 4 we just have to add 1 which means the next entry. So, ten bits means a table with thousand twenty four entries. So, 1024 microinstruction; that is the size of all the microinstruction; that is the size of the microcode sequences of all the instruction that we support. Then we can have a ten bit value for the args field, out of this 3 bits can be for the unit id. So, which unit are the arguments meant for and. So, you know the other meant for the register file or the memory unit or you know the ALU, and 7 bits for the type of operation that we want to perform. So, this part is slightly over design; nevertheless it is over designing and also useful if you want to extend the design.

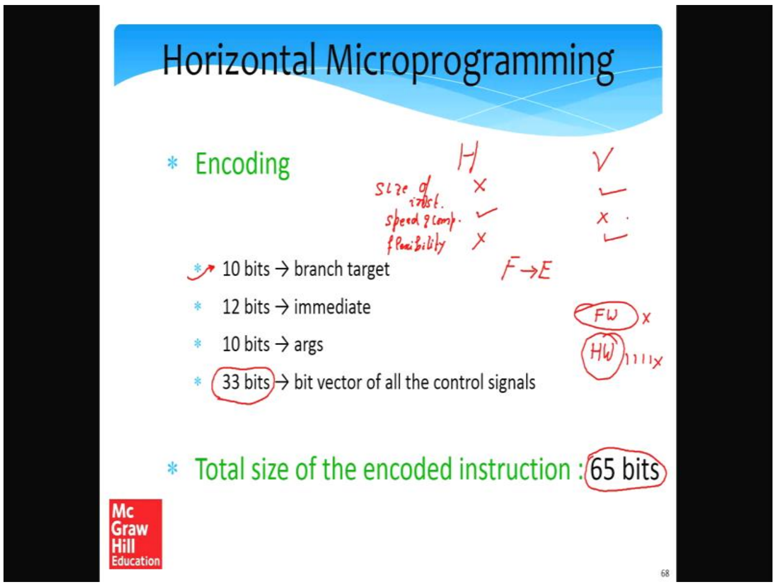
So, the total number of bits are 10 plus 10 20 plus 12 22 27 that, I am sorry 20 plus 12 is 32 37 42 and 45. This is the 45 bit instruction that we can create a 45 bit encoding, that we can create for every micro instruction. So, what is essentially required you know to process this, is a actually a processor which will have a fetch stage for the microinstruction. So, these are there in the book, then it will have the decode stage, where essentially this 45 bit is decoded. And subsequently an executes stage where the actual instruction is executed. So, it is we need a three stage processor to process this. So, I am not discussing this in detail, because they are there in the book, and also it will be a kind of reputation, because we have already looked at a lot of processing in the previous lecture. So, I do not want to talk about the same things over and over again.

Another way that we can do this is, horizontal microprogramming, where we actually get rid of the decode stage. So, you know. So, we assume that, what is decoding give us.

Decoding gives us the values of all the control signals right. Well in principle we can take all the control signals, and, because for every instruction we can compute it is control signals, and we can encode the control signals themselves. For example, let me consider an instruction of the type m add pc and 4. So, then that would may be in a, generate a long list of control signals; some zeros and some ones and so on.

So, one is that we separately encode m add separately encode the pc registers, separately encode four all of that is good, but another way can be, that we you know. Of course, we need to encode the value of four and an immediate, but for the rest we can just have a list of control signals, which will control which part of the circuit are active, and values of the read bus and write bus right, the control signals for the transfer multiplexer and so on. So, we do not need a decoding stage at all. So, this is called horizontal microprogramming, where you know the control signals will directly be part of the instruction encoding.

(Refer Slide Time: 22:16)



Horizontal Microprogramming

* Encoding
    * 10 bits → branch target
    * 12 bits → immediate
    * 10 bits → args
    * 33 bits → bit vector of all the control signals
* Total size of the encoded instruction : 65 bits

H
V
size of instr.
speed of comp.
flexibility
F → E
FW
HW

So, in this case we will have 10 bits for the branch target; fine fair enough. We will have 12 bits for the micro immediate, ten bits for the arguments, same you know in same format as this 3 bits for unit id, 7 bits for operation code. In addition you know instead of having these other fields, like type of instructions source register, you know these three fields right, we will get rid of this, instead we will have 33 bits bit vector that encodes all the control signals, where does 33 comes from.

Well, readers need to go to the books, so breakup is given. But the long and sort of it is, the summary of this is that we are getting rid of these decodes state completely, because the decoded information will be part of the instructions encoding. So, we do not need a separate decode stage. So, this will essentially be a two stage processors will have fetch and execute. So, the advantage is we are getting rid of the decode states, so our logic will be simpler, and the processing will be faster; that is the advantage of horizontal micro programming.

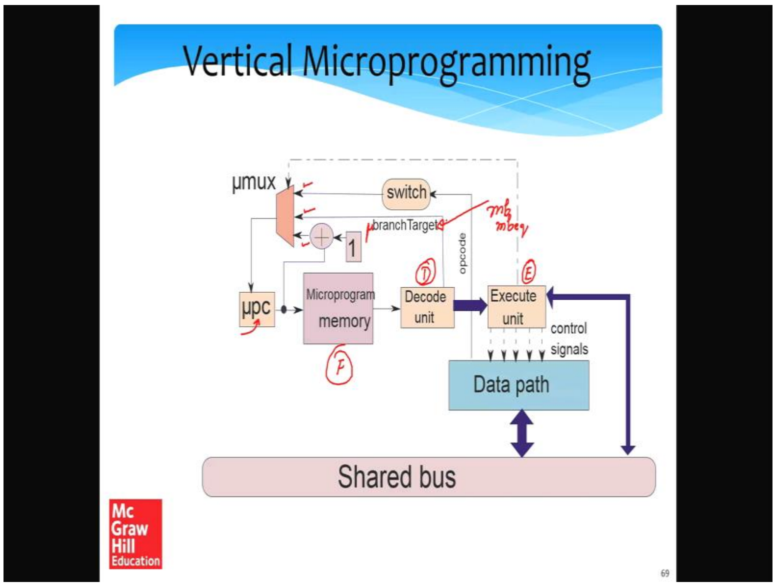
The disadvantage of horizontal microprogramming is that we will have a longer instruction, we will take one instruction, and we will require more bits to store. So, it will require 65 bits, as compared to 45 bits of vertical microprogramming. So, number one will require more size ,which when some cases, in some cases it is not, but the other bigger thing is, that we will have to expose all the details including control signals of the hardware to firmware right. So, this is hardware, and this is firmware, where you know all are microcode is there. Previously we are just disclosing the names of the micro registers and so on. So, for vertical microprogramming that was fine, but for horizontal microprogramming we will have to disclose all the control signals right, and this might be too much number one from the security point of view. Number two; we have absolutely no flexibility in changing the design even. If you change the design a little bit, these control signals will change and as a result the firmware it will become useless.

As compared to that if we use vertical microprogramming, and we still keep the register names and id is the same for example, we still have pc and i r and r s 1 r s 2 and so on, but if you make some internal changes in the control signals change, it will not affect us, because we still have a decode stage and the decode stage inside the microprocessor. Not the microprocessor the microcode unit of the processor will generate the control signals differently. So, we still have some you know freedom and flexibility.

In this case the amount of freedom and flexibility that we have is pretty much nonexistent, the reason being that we will not be able to make any change. Even if there is a mini skew change in the hardware in the control signals change. So, clearly trades off flexibility here. If I consider a horizontal microprogramming and flexible and vertical microprogramming, if I see the size of the instruction, size of inst. So, in this case; well this is bad and this is good.

If I consider may be you know a speed and complexity, and then you know this is good and this is bad. If I consider flexibility of, you know changing the control signals or other you know simple details of the hardware, horizontal has no flexibility at all. Whereas, vertical microprogramming still gives this some amount of flexibility. So, that is clearly a tradeoff between both the approaches, is not the case at one approach is strictly better than the other. So, these have to be kept in mind.

(Refer Slide Time: 26:22)

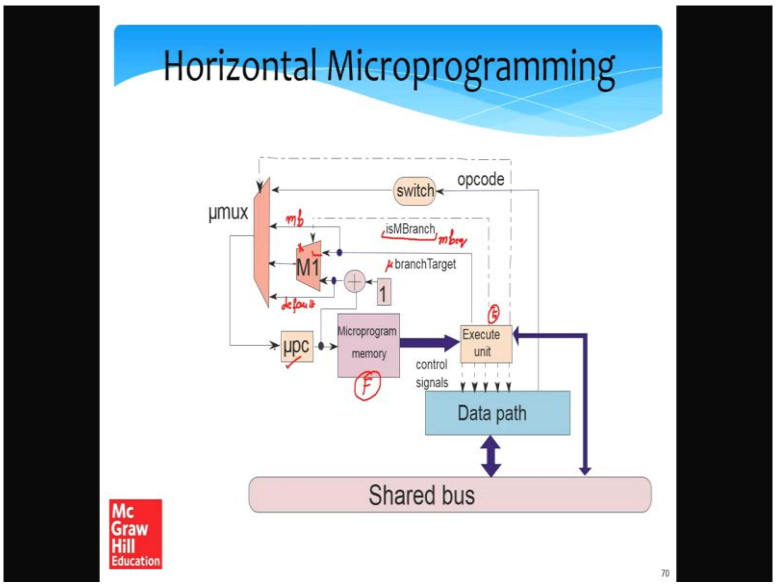


So, here is the quick description of the, you know of the microcode execution engine of a vertically microprogramming processor. So, this is very simple. So, we are not discussing this in great detail. So, we have a micro pc which goes to the micro program memory, and it basically is incremented, you know which sequentially scans it unless there is a jump. It reads each micro instructions which is send to the decode unit which enrich the control signals, that go to the execute unit. The execute unit uses the control signals to control the data path, as well as shared bus, notably the transfer multiplexer, and that is how the execution is performed. In addition the data path sends the opcode to a specialized unit called switch, which tells the micro code unit, which location in the micro program memory to jump to; that is one input of this multiplexer.

In addition we can have micro branch. So, then we can have a branch target. So, this can go here. So, this location can definitely go here, and then the default will be, that we will add one to the current micro pc, which is the next micro pc. not four, but one, because it is entry wise not byte wise. Then the micro mux will choose one of these three targets, and choose one of the micro pc. This is as such in a extremely simple processor, which basically has a fetch unit over here, which has the decode unit over here, and then the execution unit over here. And the execute unit basically let us the circuit run with the control signals that have been generated by the decode unit in the previous cycle. So, that is pretty much all; that is there to this.

I just have one clarification to make. So, there is little bit of confusion regarding the terms, but there should not be. So, this branch target is actually a micro branch target. So, maybe even I will update this in other version of the slides, so slides are there on my website. So, this should actually be the micro branch target, it is not the simple risk branch target, rather it is a branch target in the micro program memory, and this will essentially come into being, because of the mb and the mb q instructions. So, they will you know make the micro code jump from one place to the other. So, it is that branch target, it is not the simple risk branch target that is needed to be kept in mind.

(Refer Slide Time: 29:16)



So, this is the processor design of horizontal microprogramming, which is equally simple, but it is, well it is slightly simpler. So, the most of the circuits remains the same we still have a micro pc over here, which supplies the value to the micro program memory. And since all the control signals are already embedded inside the micro instruction itself, we just have a fetch stage and an execute stage. The control signals r send to the data path, where they control the movement of data, and also the transfer multiplexer. In addition you know we have actually two multiplexers here. So, this multiplexer essentially gets the micro branch target. This should be the micro branch target, and because, so that we do not confuse it with the branch target micro register that we have, which is actually for simple risk. So, this is the micro branch target, and also the default next micro program. So, we choose one of them, and that is one.

And in addition what we do, is the. So, basically you know the way that we choose. So, I will just tell you why we have two multiplexer here. So, we have 1 m 1. So, essentially if the instruction is the branch, and if the branch is successful if it is taken, then we will choose one of these. Otherwise you know if this instruction is not a branch, and it is not relevant. Then essentially we move the both the branch targets, and the default next micro program pc right to the micro mux. So, this part could have been designed differently, but the main sprit of these two multiplexer is, that we will use ism branch signal mainly for comparing with an immediate, which is basically m beq.

With the m beq micro instruction we compare the value of register with an immediate. So, this will only take care of that, other than that for a we will use the micro branch target, let say mb kind of instruction, that will directly use in the micro mux, and this will serve as the default right, which is the default next micro program counter. In the addition we will have, we will send the opcode switch unit which will generate address in the micro program memory that we need to jump, to for a given separator instruction.

So, we need to choose among these four inputs. Of course, in theory, the m one mux could have been removed, and we could have just add the micro branch target and the default value right. Possible could have been done which is sorted to meet the control circuit, slightly more complicated this was slightly simpler to design and visualize; that is the reason this design was chosen.

And you know there is no other reason, there no other reason why this cannot be removed, is just the logic has to be changed a little bit, but it is still fine, and once we choose a next pc, this can be fed to the micro program counter; the next micro pc that is we can send it to the micro program counter all right. So, this brings us to an end of the chapter on microprogramming, and we will then look at making processors and processor design much more efficiently. So, we will discuss a technique called pipelining in a next chapter which is chapter 9.