Pipelined processors overlap instructions in time on common execution resources.
I like to start today’s lecture with a look back at the single cycle processor. The reason is that this single cycle processor has a lot of common with the pipeline processor we are going to talk about today. Everything in this single cycle processor starts at the Instruction Fetch Unit which sends the Rs, Rt, Rd, and Immediate fields of the instruction to the datapath. At the same time, the Opcode and Func fields of the instruction are sent to the Control Unit. Based on the OP field of the instruction, the Main Control will set the control signals RegDst, ALUSrc, .... etc. properly. The ALU Control then uses the ALUop from the Main control and the Func field of the instruction to generate the ALUctr signals to ask the ALU to do the right thing. As far as the datapath is concerned, the data pretty much flow from left to right.

\[ +1 = 1 \text{ min. (X:41)} \]

**Question:** Why do we say that single cycle processor has more commonplaces with pipeline processor?
One of the biggest disadvantage of the single cycle implementation is that the cycle time must be long enough for the load instruction.

Having a long cycle time is a big problem but not the the only problem.
Another problem is that this cycle time (point to the list), which is long enough for the load instruction, is too long for all other instructions.

+1 = 2 min. (X:42)
Overview of a Multiple Cycle Implementation

- The root of the single cycle processor's problems:
  - The cycle time has to be long enough for the slowest instruction

- Solution:
  - Break the instruction into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
    - Cycle time: time it takes to execute the longest step
    - Keep all the steps to have similar length
  - This is the essence of the multiple cycle processor

- The advantages of the multiple cycle processor:
  - Cycle time is much shorter
  - Different instructions take different number of cycles to complete
    - Load takes five cycles
    - Jump only takes three cycles
  - Allows a functional unit to be used more than once per instruction

This (the root of these problems) leads us to the multiple cycle design where the instruction is broken into smaller steps.

And instead of executing an entire instruction in one cycle, we will execute each of these steps in one cycle.

Since the cycle time in this case will be the time it takes to execute the longest step, our goal should be to keep all the steps to have similar length when we break up the instruction.

The first advantage of the multiple cycle processor is of course shorter cycle time. The cycle time now only has to be long enough to execute the longest step.

But maybe more importantly, now different instructions can take different number of cycles to complete. For example:

1. The load instruction will take five cycles to complete.
2. But the Jump instruction will only take three cycles.

This feature greatly reduce the idle time inside the processor.

Finally, the multiple cycle implementation allows a functional unit to be used more than once per instruction as long as it is used on different clock cycles.

+2 = 4 min. (X:44)
And here it is the datapath of the multiple cycle processor.

As we said earlier, one advantage of this Multiple Cycle implementation is that a functional unit can be used more than once per instruction AS LONG it is used at different cycles.

For example, the ALU here is used for calculating the next PC as well as all the others ALU operations and the Ideal Memory here is used for storing the instruction as well as data.

The explicit price we pay for using a functional unit more than once per instruction are all the extra MUXes we need in the datapath.

The implicit price we pay is lower performance because as I will show you today, by NOT using a functional unit more than once per instruction, we can pipeline the instructions and achieve MUCH higher performance.

\[ +1 = 45 \text{ min. (X:45)} \]
Here is the outline of today’s lecture.
In the next 20 minutes or so, I will give you an overview of the pipelining idea.
Then I will show you the pipelined datapath as well as the pipelined control.
One problem will come up again is the potential race condition between the address lines and the write enable line of data memory and register file. I will show you how to avoid that.
I will also show you how the pipelined datapath works with multiple instructions in it.
Finally, I will summarize the differences between the single cycle, multiple cycle and pipeline implementations of the processor.

+1 = 6 min. (X:46)
Well let's take a look at the Load instruction's timing diagram and see how we can break it up into smaller steps. The biggest contributors to the cycle time appears to be:

1. Instruction Memory Access Time.
2. Delay through the Control Logic, which happens in parallel with Register File Access.
3. ALU Delay.
5. And Register File Write Time.

Therefore, it makes sense to break up the Load instructions into these five steps:

1. Instruction Fetch.
2. Instruction Decode “slash” Register Fetch.
3. Memory Address Calculation.
4. Data Memory Access.
5. And finally, Register File Write.

Notice that here I have used the term Register File Write time instead of Register File Write Setup time. The reason is that in a “real” register file, there is no such thing as set up time.

\[ +2 = 13 \text{ min.} \] (X:53)
The Five Stages of Load

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
</tr>
</tbody>
</table>

° Ifetch: Instruction Fetch
  • Fetch the instruction from the Instruction Memory

° Reg/Dec: Registers Fetch and Instruction Decode

° Exec: Calculate the memory address

° Mem: Read the data from the Data Memory

° Wr: Write the data back to the register file

As shown here, each of these five steps will take one clock cycle to complete.
And in pipeline terminology, each step is referred to as one stage of the pipeline.

+1 = 8 min. (X:48)
Key Ideas Behind Pipelining

- Grading the mid term exams:
  - 5 problems, five people grading the exam
  - Each person ONLY grades one problem
  - Pass the exam to the next person as soon as one finishes his part
  - Assume each problem takes 0.5 hour to grade
    - Each individual exam still takes 2.5 hours to grade
    - But with 5 people, all exams can be graded much quicker

- The load instruction has 5 stages:
  - Five independent functional units to work on each stage
    - Each functional unit is used only once
  - The 2nd load can start as soon as the 1st finishes its Ifet stage
  - Each load still takes five cycles to complete
  - The throughput, however, is much higher

Let me see whether I can give you an overview of how pipeline work with an analogy.

In the mid-term exam you just had, there were five problems. For the sake of simplicity, let’s say there are five of us who have to grade this big pile of exams.

The most efficient way to do this is to assign one problem to each person so EACH person ONLY has to grade one particular problem and be very efficient at it.

Now as soon as that person finish grading the problem assigned to him, he can just pass the exam to the next person so the next problem can be graded.

Assume each problem takes half an hour to grade, then each individual exam still takes 2.5 hours to grade but with 5 people working together, the big pile of exams can be done much quicker than if they have to be graded by a single person.

Similarly, the load instruction has five stages. So if the datapath has 5 independent functional units that are specialized to work on each stage, then as soon as the first load instruction finishes its first stage, we can start the second load instruction.

If you look at one Load instruction at a time, you will notice that each Load still takes five cycles to complete in the pipeline processor, just like the multiple cycle processor.

However, if you look at the big picture, you will notice that over the same period of time, the pipeline processor can complete many more instructions (throughput) than the multiple cycle processor because the pipeline processor will start working on the next instruction before the previous one finishes.

Let me show you what I mean by using a pipeline diagram.

\[ +3 = 11 \text{ min. (X:51)} \]
For the load instructions, the five independent functional units in the pipeline datapath are:

(a) Instruction Memory for the Ifetch stage.
(b) Register File’s Read ports for the Reg/Decode stage.
(c) ALU for the Exec stage.
(d) Data memory for the Mem stage.
(e) And finally Register File’s write port for the Write Back stage.

Notice that I have treated Register File’s read and write ports as separate functional units because the register file we have allows us to read and write at the same time.

Notice that as soon as the 1st load finishes its Ifetch stage, it no longer needs the Instruction Memory. Consequently, the 2nd load can start using the Instruction Memory (2nd Ifetch).

Furthermore, since each functional unit is only used ONCE per instruction, we will not have any conflict down the pipeline (Exec-Ifet, Mem-Exec, Wr-Mem) either.

I will show you the interaction between instructions in the pipelined datapath later. But for now, I want to point out the performance advantages of pipelining.

If these 3 load instructions are to be executed by the multiple cycle processor, it will take 15 cycles. But with pipelining, it only takes 7 cycles. This (7 cycles), however, is not the best way to look at the performance advantages of pipelining.

A better way to look at this is that we have one instruction enters the pipeline every cycle so we will have one instruction coming out of the pipeline (Wr stages) every cycle.

Consequently, the “effective” (or average) number of cycles per instruction is now ONE even though it takes a total of 5 cycles to complete each instruction.

\[ +3 = 14 \text{ min. (X:54)} \]
Well, so far so good. Let’s take a look at the R-type instructions.
The R-type instruction does NOT access data memory so it only takes four clock cycles, or in our new pipeline terminology, four stages to complete.
The Ifetch and Reg/Dec stages are identical to the Load instructions. Well they have to be because at this point, we do not know we have a R-type instruction yet.
Instead of calculating the effective address during the Exec stage, the R-type instruction will use the ALU to operate on the register operands.
The result of this ALU operation is written back to the register file during the Wr back stage.

\[+1 = 15 \text{ min. (55)}\]
What happened if we try to pipeline the R-type instructions with the Load instructions?
Well, we have a problem here!!!
We end up having two instructions trying to write to the register file at the same time!
Why do we have this problem (the write “bubble”)?
Well, the reason for this problem is that there is something I forget to tell you.

+1 = 16 min. (X:56)
I already told you that in order for pipeline to work perfectly, each functional unit can ONLY be used once per instruction.

What I have not told you is that this (1st bullet) is a necessary but NOT sufficient condition for pipeline to work.

The other condition to prevent pipeline hiccup is that each functional unit must be used at the same stage for all instructions.

For example here, the load instruction uses the Register File’s Wr port during its 5th stage but the R-type instruction right now will use the Register File’s port during its 4th stage.

This (5 versus 4) is what caused our problem. How do we solve it? We have 2 solutions.

\[ +1 = 17 \min \] (X:57)
The first solution is to insert a “bubble” into the pipeline AFTER the load instruction to push back every instruction that are already in the pipeline by one cycle.

At the same time, the bubble will delay the Instruction Fetch of the instruction that is about to enter the pipeline by one cycle.

Needless to say, the control logic to accomplish this can be complex.

Furthermore, this solution also has a negative impact on performance.

Notice that due to the “extra” stage (Mem) Load instruction has, we will not have one instruction finishes every cycle (points to Cycle 5).

Consequently, a mix of load and R-type instruction will NOT have an average CPI of 1 because in effect, the Load instruction has an effective CPI of 2.

So this is not that hot an idea. Let’s try something else.

\[ +2 = 19 \text{ min. (X:59)} \]
Well one thing we can do is to add a “Nop” stage to the R-type instruction pipeline to delay its register file write by one cycle.

Now the R-type instruction ALSO uses the register file’s write port at its 5th stage so we eliminate the write conflict with the load instruction.

This is a much simpler solution as far as the control logic is concerned. As far as performance is concerned, we also get back to having one instruction completes per cycle.

This is kind of like promoting socialism: by making each individual R-type instruction takes 5 cycles instead of 4 cycles to finish, our overall performance is actually better off.

The reason for this higher performance is that we end up having a more efficient pipeline.

\[ +1 = 20 \text{ min. (Y:00)} \]
Let's continue our lecture by looking at the store instruction.

Once again, the Ifetch and Reg/Decode stages are the same as all other instructions. The Exec stage of the store instruction calculates the memory address. Once the address is calculated, the store instruction will write the data it read from the register file back at the Reg/Decode stage into the data memory during the Mem stage.

Notice that unlike the load instruction which takes five cycles to accomplish its task, the Store instruction only takes four cycles or four pipe stages.

In order to keep our pipeline diagram looks more uniform, however, we will keep the Wr stage for the store instruction in the pipeline diagram.

But keep in mind that as far as the pipelined control and pipelined datapath are concerned, the store instruction requires NOTHING to be done once it finishes its Mem stage.

\[ +2 = 27 \text{ min. (Y:07)} \]
Well similar to the store instruction, the branch instruction only consists of four pipe stages. Ifetch and Reg/decode are the same as all other instructions because we do not know what instruction we have at this point. We have not finish decoding the instruction yet.

During the Execute stage of the pipeline, the BEQ instruction will use the ALU to compare the two register operands it fetched during the Reg/Dec stage.

At the same time, a separate adder is used to calculate the branch target address.

If the registers we compared during the Execute stage (point to the last bullet) have the same value, the branch is taken.

That is, the branch target address we calculated earlier (last bullet) will be written into the Program Counter.

With all these talk about pipelined datapath and pipelined control, let’s take a look at how the pipelined datapath looks like.

\[ +2 = 29 \text{ min. (Y:09)} \]
The pipelined datapath consists of combination logic blocks separated by pipeline registers. If you get rid of all these registers (not the PC), this pipelined datapath is reduced to the single-cycle datapath.

This should give you extra incentive to do a good job on your single cycle processor design homework because you can build your pipeline design based on your single cycle design.

Anyway, the registers mark the beginning and the end of a pipe stage.

In the multiple clock cycle lecture, I recommended that the best way to think about a logic clock cycle is that it begins slightly after the clock tick and ends right at the next clock tick.

For example here, the Reg/Decode stage begins slightly after this clock tick when the output of the IF/ID register has stabilized to its new value AND ends RIGHT at the next clock tick when the output of the register file is clocked into the ID/Exec register.

At the end of the Reg/Decode stage, the register output that just clocked into the ID/Exec register has NOT yet propagate to the register output yet. It takes a Clk-to-Q delay.

When the new value we just clocked in (points to the clock tick) has propagate to the register output, then we have reach the beginning of the Exec stage.

Notice that the Wr stage of the pipeline starts here (last cycle) but there is no corresponding datapath underneath it because the Wr stage of the pipeline is handled by the same part of the pipeline that handles the Register Read stage.

This part of the datapath (Reg File) is the only part that is used by more than one stage of the pipeline. This is OK because the register file has independent Read and Write ports.

More specifically, the Reg/Decode stage of the pipeline uses the register file’s read port while the Write Back stage of the pipeline uses the register file’s write port.

+3 = 32 min. (Y:12)
Let’s follow a Load instruction down the pipeline and see how everything works.

Every instruction starts at the Instruction Fetch stage which:
(a) Begins slightly after the first clock tick when PC output has stabilized to its new value.
(b) And ends when the output of the I-Unit is clocked into the IF/ID register.

This picture shows the state of the pipeline at the end of the Ifetch stage (you are here).
Let’s expand this part of the datapath and take a better look.

\[ +1 = 33 \text{ min. (Y:13)} \]
Let's assume the Load instruction we are following is in memory location TEN, 10.

So the Ifetch stage begins shortly after the first clock tick when the value 10 appears on the Program Counter register output.

This is used as the address to access the Instruction Memory and at the same time it is fed to the adder so it can be increment by four.

At the end of the Ifetch stage, this clock tick will clock the output of the instruction memory, which is the Load instruction, into the If/ID pipeline register.

As the same time, the PC plus 4 value, that is 14 in this case, will be clocked into the PC.

Notice that the picture here shows the stage of the pipeline at the END of the Ifetch stage (you are here) so the number 14, even though is is already latched into the register, will not appear at the PC output until Clk-to-Q time later.

Similarly, the load instruction we just clocked into the IF/ID register will NOT appear at the register output until a Clk-toQ delay after this clock.

And when it appears at the IF/ID register output, we will be in the Reg/Decode stage.

+2 = 35 min. (Y:15)
This picture shows the state of the pipeline at the end of Load's Reg/Decode stage. All the datapath has to do is read register (Ra and busA) R2 from the register file. As the same time, we need to pass the Immediate field of the instruction onto the next stage (ID/Exec) register because the immediate field is needed for address calculation.

Let's concentrate on the load instruction (point to the 1st bullet) going down the pipe and ignore what is in the PC and IF/ID registers for now. Since we are here (Your are Here), Register R2 and Imm16 have already clocked into ID/Exec register but they have not yet appeared at the register’s output.

\[ +1 = 36 \text{ min. (Y:16)} \]
If we wait a little bit longer (beginning of Exe), Register R2 and Imm16 will appear at the ID/Exec register output and we are now in the Exec stage where we will calculate the Memory Address for the load instruction.

Notice that besides calculating the Load's Memory Address, we also need to pass the Rt field (RegDst = 0) of the instruction down the pipeline.

The reason is that we will need to use Rt field later on to specify which register to write to when we reach the end of the Load pipeline.

Also notice that this is the first stage of the pipeline that requires any control signals.

We will talk more about this (Rt field and other control signals) later but for now, let's expand this part of the datapath and take a closer look at what is going on.

+2 = 38 min. (Y:18)
At the beginning of the Exec cycle, ID/Exec register’s output will be stabilized to Register R2 (bus A) and Imm16.

By setting the control signals ExtOp and ALUSrc to 1s, the 16-bit immediate field will be sign extended to 32 bits and pass onto the ALU.

The ALU then add (ALUOp = Add) this sign extended version of the Immediate field to Register 2 (busA) to form the memory address.

By the end of the Exec cycle (you are here), the ALU output will be valid and will be stored into the Exec/Mem pipeline register.

This Memory Address (Exec/Mem) will appear at the output of the Exec/Mem register a Clk-to-Q time after this clock tick (you are here).

+1 = 39 min. (Y:19)
And it will be used to read the Data Memory (RA).

By setting the clock cycle time longer than the data memory read access time, we can guarantee the memory output (Do) will be stable by the end of the Mem cycle (you are here).

And this clock tick will trigger the Mem/Wr register to latch in Load’s data.

We have to pass the Load instruction’s Rt field down the pipeline one more time because we still have not use it yet.

\[ +1 = 40 \text{ min. (Y:20)} \]
The Rt filed of the load instruction will be used in the Register Write back stage of the pipeline (Wr) to specified the register (Rw) where the load data is written.

Since we are writing the load’s data into the register file, the control signal MemtoReg and RegWr must be set to 1.

So this finishes the grand tour of how a load instruction flows through the datapath.

But how about control signals?

\[ +1 = 41 \text{ min. (Y:21)} \]
The key observation about the control signals is that the control signals at Stage N depends ONLY on the instruction that is currently in Stage N, where N is either Exec, Mem, or Wr (points to the control signals at each stage, especially RegWr at Wr stage).

Notice that I did not mention anything about the Ifet and Dec stage because there cannot be any instruction dependent control signals at these two stages. Why?

Because at these two stages (Ifetch and Dec) we do not know what instructions we have yet: we have just fetch the instruction (Ifetch) and is still in the process of decoding it (Dec).

Here is an example of this rule (1st bullet). During the Load instruction’s Exec stage, the control signals (ExtOp, ALUSrc, ALUOp) in the Exec stage are set this way because this is the setting required to do the right things for Load instruction's Exec stage.

Since the control signals depend ONLY on the instruction that is currently in that stage of the pipeline (1st Bullet), the control is very similar to the Controller for the single cycle processor.

The difference is that after we generate the control signals, we need to pipeline them so they are applied to the correct stage of the pipeline (Exec, Mem, an Wr) when the instruction reaches that stage.

Let me show you what I mean by this.

\[ +2 = 43 \text{ min.} \]
The main control here is identical to the one in the single cycle processor. It generates all the control signals necessary for a given instruction during that instruction’s Reg/Decode stage.

All these control signals will be saved in the ID/Exec pipeline register at the end of the Reg/Decode cycle.

The control signals for the Exec stage (ALUSrc, ...) come from the output of the ID/Exec register. That is they are delayed ONE cycle from the cycle they are generated.

The rest of the control signals that are not used during the Exec stage is passed down the pipeline and saved in the Exec/Mem register.

The control signals for the Mem stage (MemWr, Branch) come from the output of the Exec/Mem register. That is they are delayed two cycles from the cycle they are generated.

Finally, the control signals for the Wr stage (MemtoReg & RegWr) come from the output of the Exec/Wr register: they are delayed three cycles from the cycle they are generated.

\[ +2 = 45 \text{ min. (Y:45)} \]
Let's focus our attention to the beginning of the Wr stage and see what happens there.

Recalled from our pipelined datapath that at the beginning of the Register Write back cycle, the register address, that is Rw, will come from the output of the Mem/Wr pipeline register.

From our pipeline control discussion, we also know that the register write enable signal (RegWr) will also come from the pipeline register Mem/Wr.

Consequently, at the beginning of the Register Wr stage, we can have a problem if the RegAdr's Clock-to-Q time is longer than the RegWr's clock to Q time.

Because in that case, RegWr can be asserted BEFORE the register specifier (RegAdr) is settled to its final value and cause us to write to the wrong register.

Similarly, we can have a problem at the beginning of the Memory Access stage if the Clock-to-Q time of the memory Write address is longer than the MemWr's clock-to-q time.

In either case, we have a race condition between the address lines and the write enable control signal.

Did anybody remember how did we prevent this (address and write enable) race condition in our multiple cycle design?

\[ +2 = 52 \text{ min. (Y:32)} \]
The Pipeline Problem

- Multiple Cycle design prevents race condition between Addr and WrEn:
  - Make sure the address is stable by the end of Cycle N
  - Asserts WrEn during Cycle N + 1

- This approach cannot be used in the pipeline design because:
  - Must be able to write the register file every cycle
  - Must be able to write the data memory every cycle

Solution? Recall 1-cycle processor’s approach.

Well in our multiple cycle design, we prevent the race condition between the write address lines and the write enable signal by:

(a) Making sure the address is stable by the end of a given cycle, say Cycle N.
(b) Then we assert the write enable line to write the memory 1 cycle later (Cycle N + 1).

In other words, it will take us two cycles (Cycle N and Cycle N + 1) to write anything.

This approach, however, cannot be used in the pipeline design because we must be able to write the register file every cycle when we have consecutive R-type instructions.

Similarly, we must be able to write the data memory every cycle when we encounter consecutive store instructions.

So we need to look for a solution that is different from the multiple cycle implementation.

+2 = 54 min. (Y:34)
The solution we have is to AND the write enable signal to the clock input.

The Write Enable signal to register file or memory is probably the ONLY place where you, as a logic designer, can gate (AND) the clock with a control signal.

This is the place where you need to take off your logic designer’s hat and put on an electrical engineer’s hat to make sure by ANDING the write enable signal to the clock signal, you are NOT violating any timing consideration.

One obvious thing you should check is that the clock high time must be greater than the write access time because the C_WrEn signal will only be high during the clock high time.

After the circuit designer has taken a good look at this block and made sure no timing constraint is violated, he or she will create a symbol like this and give it to the logic designer.

And he probably will tell the logic designer something like: “Hey, don’t change anything in this block without first asking me.” The logic designer can then use it just like any logic element.

What we created here (the box) is what in the industry is called Synchronized SRAM if this block is the Data Memory. Or Synchronized Register File if this block is the Register File.

The synchronized memory and register file should be no stranger to you guys because I have already shown you how they can be used in the Single Cycle Datapath lecture.

All you have to do is make sure the address, data, and write enable signal are stable at least one set-up time before the clock tick.

The actual write will occur at the cycle following the clock edge that captures the WrEn=1.

+3 = 57 min. (Y:37)
Well, let’s look at a more complex example so I can show you how different instructions at different stages of execution can be processed by our pipelined datapath simultaneously.

Let’s consider the following instruction sequence: Load, R-type, Store, and then Branch on equal to target address 1000.

In the next four slides, I will show you the state of the datapath at the end of Cycle 4, Cycle 5, Cycle 6 and Cycle 7.

First let’s take a look at the end of Cycle 4 where:
(a) The Load instruction has just finished its Mem stage.
(b) The R-type instruction has just finished its Exec stage.
(c) The Store instruction has just finished its Register Fetch slash Instruction Decode stage.
(d) And finally, the Branch instruction has just finish fetching the instruction.

Remember now, the next four pictures we will be looking at are at the end of a clock cycle. That is right at the clock tick.

+2 = 59 min. (Y:39)
So the internal value of the register has just been updated BUT the output of the register, due to the Clock-to-Q delay, has NOT been changed to reflect the new internal value.

The output of the register will still have the value from the cycle that has just ended.

For example, look at Ifetch stage of the pipeline. The output of PC will still have the values of 12, the location of the Branch instruction even though PC has been updated to 16.

Furthermore, the IF/ID register’s internal value have already been updated to the Branch instruction even though its output still contains the Store instruction so that we can read the registers (Ra and Rb) for the store instruction and store their values into the ID/Ex register.

In order to get ready for calculating the store address, we also needs to pass the immediate field to the next stage.

At the end of Cycle 4, the Exec stage of the pipeline have just completed the ALUOp for the R-type instruction so the ALU output is stored into the Exec/Mem register.

Notice that for R-type instruction, the Rd field of the instruction is used to specify the destination register so the RegDst control signal must be set to 1 to pass Rd down the pipe.

The Mem stage of the pipeline have just completed the data memory read access for the load instruction so the data to be loaded can be found in the Mem/Wr register.

In the pipeline diagram, I did not show any instruction prior to the load instruction so we do NOT have any instruction in the Wr stage yet (MemtoReg=X, RegWr=0).

So far so good. Let’s take a look at what happened at the next clock cycle.

\[ +3 = 62 \text{ min. (Y:42)} \]
Well we are now at the END of Cycle 5. Looking at Ifetch stage of the pipeline. The output of PC will still have the values of 16 even though PC has been updated to 20.

I have assumed the instruction at memory location 16 to be a R-type instruction so the IF/ID register’s internal value will contain this R-type instruction even though its output still contains the Branch instruction that has just completed its Register Fetch stage (ID/Ex).

In order to get ready for calculating the branch target address, we also needs to pass the immediate field to the next stage.

At the end of Cycle 5, the Exec stage of the pipeline has just completed the address calculation for the Store instruction which involves adding the sign extended (ExtOp=1) version of the Immediate field (ALUSrcA) to the Rs register (busA).

This Store address is stored in the Ex/Mem register so it can be used in the next clock cycle.

Notice that for R-type instruction, the Mem stage is a NoOp stage. All we do is to pass the ALU output AS WELL AS the Rd field from the ID/Ex register to the Mem/Wr register.

At the end of Cycle 5, the Wr stage of the pipeline has just completed the Register File write for the load instruction.

Therefor, RegWr is set to 1 for the register write and MemtoReg is set to 1 to route the data memory output we saved from last cycle (output of Mem/Wr register) to the register file.

+3 = 65 min. (Y:45)
Now, let take a look at the end of Cycle 6.

The Branch instruction has just completed its Exec stage so the signal Zero will be asserted if the two registers (busA and busB) we compared (ALUOp=sub) are equal.

This value of Zero (2nd from top) as well as the branch target address (top), that is the sum of PC+4 and the sign extended version (ExtOp=1) of the immediate filed, must be stored in the Ex/Mem register (Beq’s result).

The Mem stage of the pipeline has just completed the operation for the store instruction where the value from bus B is stored (MemWr=1) into the memory address location specified by the ALU output.

The Wr stage of the pipeline has just completed the operation for the R-type instruction.

Therefore MemtoReg is set to zero to route the ALU output we saved in the Mem/Reg register to the register file where RegWrite is set to 1 to complete the write.

\[ +2 = 47 \text{ min. (Y:47)} \]
Finally, let's take a look at the end of Cycle 7 where the Branch instruction has just completed its Mem stage.

I have assumed the registers we compared in the last cycle are indeed equal so ALU output Zero was asserted and captured in the Ex/Mem register.

With Zero asserted and the control signal Branch set to 1, the branch target address we calculated in Cycle 6 and saved in the Ex/Mem register is now written into the PC (1000).

By writing the branch target address into the Program Counter, we have just completed the branch instruction.

\[ +2 = 69 \text{ min. (Y:49)} \]
The Delay Branch Phenomenon

Although Beq is fetched during Cycle 4:
- Target address is NOT written into the PC until the end of Cycle 7
- Branch's target is NOT fetched until Cycle 8
- 3-instruction delay before the branch take effect

This is referred to as Branch Hazard:
- Clever design techniques can reduce the delay to ONE instruction

Notice that although the branch instruction is fetched during Cycle 4, its target address is NOT written into the Program Counter until the end of clock Cycle 7.

Consequently, the branch target instruction is not fetched until clock Cycle 8.
In other words, there is a 3-instruction delay between the branch instruction is issued and the branch effect is felt in the program.
This is referred to as Branch Hazard in the text book.
And as we will show in the next lecture, by clever design techniques, we can reduce the delay to ONE instruction.
That is if the branch instruction in Location 12 is issued in Cycle 4, we will only execute one more sequential instruction (Location 16) before the branch target is executed.

+2 = 71 min. (Y:51)
Similarly, although the load instruction is fetched during cycle 1, the data is not written into the register file until the end of Cycle 5. Consequently, the earliest time we can read this value from the register file is in Cycle 6. In other words, there is a 3-instruction delay before the load takes effect.

This is referred to as Data Hazard:
- Clever design techniques can reduce the delay to ONE instruction.

+2 = 73 min. (Y:53)
Let me summarized the different processor implementations we have learned in the last few lectures.

First the single cycle processor has two disadvantages: (1) long cycle time. (2) And may be more importantly, this long cycle time is too long for all instructions except the load.

The multiple clock cycle processor solve these problems by:
(1) Dividing the instructions into smaller steps.
(2) Then execute each step (instead of the entire instruction) in one clock cycle.

The pipeline processor is just a natural extension of the multiple clock cycle processor.

In order to convert the multiple clock cycle processor into a pipeline processor, we need to make sure each functional unit is only used once per instruction. Furthermore, if a instruction is going to use a functional unit, it must use it at the SAME stage as all other instructions.

Finally, for the pipeline control, the most important thing you need to remember is that the control signals at each stage depends ONLY on the instruction that is currently in that stage.

+2 = 75 min. (Y:55)
Here are the timing diagrams showing the differences between the single cycle, multiple cycle, and pipeline implementations.

For example, in the pipeline implementation, we can finish executing the Load, Store, and R-type instruction sequence in seven cycles.

In the multiple clock cycle implementation, however, we cannot start executing the store until Cycle 6 because we must wait for the load instruction to complete.

Similarly, we cannot start the execution of the R-type instruction until the store instruction has completed its execution in Cycle 9.

In the Single Cycle implementation, the cycle time is set to accommodate the longest instruction, the Load instruction.

Consequently, the cycle time for the Single Cycle implementation can be five times longer than the multiple cycle implementation.

But may be more importantly, since the cycle time has to be long enough for the load instruction, it is too long for the store instruction so the last part of the cycle here is wasted.

\[+2 = 77 \text{ min. (X:57)}\]