ECE4680

Computer Organization and Architecture

Designing a Multiple Cycle Processor

Start X:40
Let’s start today’s lecture with a review of what we did. At the end of last Friday’s lecture, we finish implementing a single cycle processor that looks like this.

The Instruction Fetch Unit gives us the instruction. The OP field is fed to the Main Control for decode and the Func field is fed to the ALU Control for local decoding.

The Rt, Rs, Rd, and Imm16 fields of the instruction are fed to the datapath.

Based on the OP field of the instruction, the Main Control of will set the control signals RegDst, ALUSrc, .... etc properly.

The ALU Control 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: Add, Subtract, and so on.

If this is a Powerview schematic of our single cycle implementation, we can do a Push and take a look at the inside of the Instruction Fetch Unit.

\[ +2 = 2 \text{ min. (X:42)} \]
Concepts

Datapath

Functions/Combinational

State/Memory/Sequential

Connections

Bus for data

Control Signals

instruction → Control → signals
The Instruction Fetch Unit looks like this. It consists of a 30-bit Program Counter, two 30-bit adders, and some Ideal Memory where instructions are stored.

The 30-bit Program Counter is used as the upper 30 bits of the address when we access the Instruction Memory.

The two least significant bits of the address are always zeros because all instructions are four bytes longs.

The two adders are used for calculating the sequential (points to plus 1) and branch (points to the output of the Sign Extender) addresses, respectively.

Let’s pop back up to our top level picture.

+1 = 3 min. (X:43)
Well, the Main Control is implemented in a rather regular structure called a PLA.
The row of AND gates decode the Opcode bits to decide what type of instructions we have.
The row of OR gates then generate the control signals based on whether a particular control signal
needs to be asserted for a given type of instruction.
For example here (1st Row), the OR gate says the control signal RegWr needs to be asserted for
R-type, Or Immediate, and Load instructions.
Well enough for the review. Let’s take a look at what we are going to learn today.

\[+1 = 5 \text{ min. (X:45)}\]
Drawbacks of this Single Cycle Processor

- Long cycle time:
  - Cycle time must be long enough for the load instruction:
    - PC’s Clock-to-Q
    - Instruction Memory Access Time
    - Register File Access Time
    - ALU Delay (address calculation)
    - Data Memory Access Time
    - Register File Setup Time
    - Clock Skew

- Cycle time is much longer than needed for all other instructions.
  Examples:
  - R-type instructions do not require data memory access
  - Jump does not require ALU operation nor data memory access

One of the biggest disadvantages of the single cycle implementation: it has a long cycle time. More specifically, the cycle time must be long enough for the load instruction which has the following seven components:

1. Clock to Q time of the Program Counter.
2. Instruction Memory Access Time.
4. ALU delay to perform a 32-bit address calculation.
5. Data Memory Access Time
6. And finally the Set Up time for Register File Write and 7. Potential Clock Skew.

Having a long cycle time is a big problem but not 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. For example:

1. The R-type instruction does not need to have a cycle time as long as the load instruction because the R-type instructions do not require any data memory access.
2. Similarly, the Jump instruction does not need a cycle time this long because the Jump does not require any data memory access nor ALU operation.

Consequently, for the R-type and Jump instruction, the processor is actually doing nothing at the last part of the clock cycle.

+2 = 8 min. (X:48)
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, not the longest instruction
    - 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
    - Adder + ALU
    - Instruction mem + Data mem

Well, the root of these problems of course is that fact that the Single Cycle Processor’s cycle time has to be long enough for the slowest instruction.

The solution is simple. Just break the instruction 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 keeping all the steps to have similar length when we break up the instruction.

Well the last two bullets pretty much summarize what a multiple cycle processor is all about.

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 may be 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.

For example, as I will show you later in today’s lecture, we can use the ALU to increment the Program Counter as well as doing address calculation.

\[+3 = 11 \text{ min. (X:51)}\]
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.

\[+2 = 13 \text{ min. (X:53)}\]
Because in a real register file, there is NO clock input (use the bottom picture).

In previous lectures, I tried to simplify things by giving both the register file and data memory a clock input such that all write happens at the clock tick—that is H to L transition of the clock.

Consequently, the address bus, the Data In bus, and the Write Enable signals must ALL be stable AT LEAST one “set-up” time before the clock tick.

But in real life, neither register file nor ideal data memory has clock input.

The Write path is pure combinational. That is after the control signal:

(1) Write Enable has gone to 1 and Din settles down
(2) Memory write access delay
(3) Before the value on the Data In bus is written into the memory location specified by the address bus.

It is very VERY important that the address bus is stable BEFORE the control signal Write Enable is set to 1.

Otherwise, you may end up destroying data already in memory by writing to the wrong address location if there is any glitches on the address bus when Write Enable is asserted.

+2 = 15 min. (X:55)
Notice that this real register file, which does not have a clock input, may not work reliably in our single cycle processor because if you look at the timing diagram, you will notice that:

1. We cannot guarantee \( R_w \), which specifies the register to be written, will be stable BEFORE \( \text{RegWr} = 1 \).
2. There is a “race” between \( R_w \) (address) and \( \text{RegWr} \) (write enable).

On a good day, if \( R_w \) does settle down before \( \text{RegWr} \) goes to 1, everything works. But once in a while, if \( \text{RegWr} \) happens to go to 1 before \( R_w \) settles down, we have a problem.

Race condition like this is what caused the machine to crash mysteriously during initial testing.

Similarly, I did not use this data memory in our single cycle processor design because we cannot guarantee the address bus to be stable BEFORE \( \text{WrEn} = 1 \).

Once again, we have a race between the Address and the Write Enable signal. How can we avoid these two race conditions in our multiple cycle implementation?

\[ +2 = 17 \text{ min. (X:57)} \]
How to Avoid this Race Condition?

Solution for the multiple cycle implementation:

- Make sure Address is stable by the end of Cycle N
- Assert Write Enable signal ONE cycle later at Cycle (N + 1)
- Address cannot change until Write Enable is disasserted

Well, for the multiple cycle implementation, we can avoid this race condition by:

1. Making sure the address bus is stable by the end of Cycle N.
2. Then we can assert the write enable signal ONE cycle later at Cycle N + 1.
3. Finally, we have to make sure the address bus does not change until the Write Enable signal is disasserted.

+1 = 18 min. (X:18)
One important feature of the multiple cycle implementation is that a functional unit can be used more than once per instruction as long as it is used on different clock cycles.

The Ideal Memory is one such unit which our multiple cycle processor will use more than once per instruction. More specifically...

Unlike the single processor which has separate Instruction and Data memory, the Multiple Cycle Implementation will only have 1 memory unit where instructions and data are stored.

In order to keep things conceptually simple, the Ideal Memory will have independent Read port (Read Address and Data Out) and Write port (Write Address and Data In).

That is, if we put an address on the Read Address inputs, the Data Out bus will be valid after the Memory Read Access Delay time.

On the other hand, if you put an address on the Write Address inputs and THEN assert the Write Enable signal, the value on the Data In bus will be written into memory location specified by the address after the Memory Write Access Delay Time.

+2 = 20 min. (Y:00)
As far as LOGIC is concerned, I think the easiest way to think about a clock cycle is that a clock cycle begins right AFTER a clock tick and ends at the next clock tick.

I have intentionally shown the L time to be much longer than the H time to emphasis a point: the H and L time does not affect your design as long as you use the simple clocking methodology where all storage elements are triggered at the same clock tick.

The only important thing here is the time between the two clock ticks, the cycle time.

Most of the time, however, the high and low time are the same because it is much easier to generate a clock that has high and low time the same length. Well enough about clock ticks.

Let’s see what happens at the beginning (You are Here) of the Ifetch cycle:

(a) We need to fetch the instruction from Memory so we sent the address to the memory.
(b) We also needs to update the PC so we better send the address to the ALU as well.

***** What values do you think the control signals PCwr and ALUop have at this point?

Well since we are only at the beginning of the cycle (Your are Here), these two signals will still have the old values from the last cycle of the previous value. See next slide of their new values.

\[ +2 = 27 \text{ min. (Y:07)} \]
As time goes by, the output of the memory will become valid and the ALU, with ALUOp sets to Add, will finish the 32-bit add.

Hopefully, we are smart enough to set the cycle time so the time between the clock tick is long enough to allow these (output of Memory and ALU) to stabilize.

So at the end of the cycle, the clock tick will trigger the Instruction Register to save the current instruction word (output of Instruction Memory).

Similarly, the Program Counter register is triggered (point to the clock input) to save the next instruction’s address (output of ALU).

Unlike the single cycle processor where a 30-bit PC can reduce the length of two adders by two bits, here we are using the 32-bit ALU to do the PC update anyway.

So the only saving we can get for using a 32 bit PC are two register bits. That’s why we didn’t bother to do it and keep a 32-bit Program Counter.

The Memory Unit here is also used to store data and the ALU here is also use for instruction execution. Therefore, we know we will need some MUXes in front of them.

+2 = 29 min. (Y:09)
For example, the Memory can get its read address from the PC for instruction fetch but it can also get the read address from other part of the datapath for data fetch.

Similarly, the ALU can get its operands from the PC and a constant 4 as I showed you on the last slide, but we know the ALU can also gets its operands from the register file.

We will fill in the details here (Hole) later but for now, we know we need to set the control signals IorD, MemWr, ALUSelA, ALUSelB to zeros and IRWr and PCWr to 1s.

Notice that I have added a MUX in the PC feedback path because we know for the branch instruction, the next PC will have a value OTHER than PC plus 4 (ALU inputs).

We will worry about how we get this “other” value (Target) later. For this cycle, we have to set the MUX control (PCSrc) to zero to select the PC plus 4 value.

The settings of all the control signals are summarized in this circle. Due to space limitation, I have only shown the signals that have values other than zeros.

I want to emphasis that this is the picture at the END of the Instruction Fetch Cycle where evaluation is completed and control signals are settled.

This is the interesting part. The start of the cycle is boring by comparison. Consequently, all the datapath pictures I show from now on are the pictures at end of a cycle.

\[ +2 = 31 \text{ min.} \ (Y:11) \]
Now that we have the instruction word saved in the IR, the next thing we can do is decode the instruction (Go to the Control) and fetch the registers from the Register file (Rs Rt).

I want to point out at this point, we do not know what instruction we have yet because we are still in the process of decoding the Op and Func field.

Therefore we are “jumping the gun” in fetching the registers Rs and Rt from the register file. The Rt field may not even be a source register if we have a I-type instruction.

But this is OK because if after we decode the instruction, we realize we don’t need the registers, we just don’t use them. No big deal.

Notice that the ALU is not being used in this cycle. That is not good. Instead of just letting the ALU sits idle, we may just as well let it do something.

***** Can we think of anyway we can use this ALU at this cycle? (see next slide)

We cannot use the ALU to do anything involving the registers because we are still in the process of reading them-we do not have the register values yet.

\[ +2 = 33 \text{ min. (X:13)} \]
What we can do is to use this ALU to calculate the branch address in advance.

(1) We will set ALUSelA to 0 such that the PC is fed to the ALU input.

(2) The other ALU input will come from (ALUSelB=10) the Sign Extended (ExtOp=1) version of the 16-bit immediate field.

Once we added (ALUOp = Add) these two numbers together, we will save the result in the Target register (BrWr = 1). We cannot write the ALU output to the PC yet (PCWr = 0).

The OP and Func is still being decode in this cycle (Control) so we cannot update the PC to this value unless we are SURE we have a branch and the branch condition is met (AND-OR).

Once again, I have summarized all the control signals settings inside this circle. So far this and the Instruction Fetch cycles are shared by all instructions.

But by the end of this cycle, we will know exactly what instruction we have (Control output). Let’s say, we have a branch, what do we do?

\[ +2 = 35 \text{ min. (Y:15)} \]
We already have the values of registers Rs and Rt on busA and busB from last cycle, all we have to do is perform a Subtract (ALUOp) to compare them (ALUSelA, B).

If they are equal, the ALU’s Zero output will be asserted, and with PCSrc and PCWrCond set to one, the Branch Target will get written into the Program Counter. The Branch is taken.

If Rs and Rt are not equal, Zero will not be asserted and the Target value will NOT be written into the Program Counter. That is the Branch is NOT taken.

Since I am running out of space in this circle, I did not say it explicitly but all control signals not specified in this circle are default to zeros (point to the datapath for examples).

+1 = 36 min. (Y:16)
Let's go back to the end of the Register Fetch slash Instruction Decode Cycle.
Assume the result of the Instruction Decode indicates we have a R-type instruction, what do we do then? Well, simple enough: we just go to the R-type execution cycle.

+1 = 37 min. (Y:17)
Once again, fetching the registers Rs and Rt in the previous cycle pays off. We need these two registers now and they are already on busA and busB, respectively.

So all we need is set the ALUSelA and ALUSelB to feed busA and busB into the ALU and tell the ALU local control we have a R-type instruction (ALUOp).

The ALU will then generate the correct result (ALU output) at the end of this cycle.

Notice that I have set RegDst to 1 here even though we are not writing the register file (RegWr is zero). Register file is not written until the next cycle.

You would think RegDst should be don’t care at this point.

****** Anybody want to guess why I set RegDst to 1 at this point?

Remember: for this Real memory and register file that do not have a clock input, the address (Rw) MUST be stable BEFORE we set Write Enable (RegWr) to one.

Here by setting RegDst to one, I can guarantee the Rw specifier will be stable by the next clock cycle where I will perform the write by setting RegWr to 1.

+2 = 39 min. (Y:19)
So here is the picture where we finish off the R-type instruction by writing the ALU output back to the register file (MemtoReg = 0 and RegWr = 1).

Notice that in order to keep the ALU output from changing, the ALUSelA, ALUSelB, and ALUOp control signals must remain the same as the previous cycle.

This brings us to a side topic I want to cover.

+1 = 40 min. (Y:20)
The side topic I want to cover in the next few minutes is called Multiple Cycle Delay Path. Looking back at the Register Fetch, R-type Execution, and R-type Completion cycles, you will notice that there is no registers in between the cycles to save the results.

That is, at the end of the Register Fetch cycle, registers Rs and Rt are placed on busA and busB but we do not have a registers on busA and busB to save them. They just sit on the bus and wait until the next cycle, the R-type Execution cycle, when they propagate through the MUXes and into the ALU inputs.

By the end of the R-type Execution cycle, the ALU output is valid and once again, the value is not saved in a register. It just sits there and wait until the R-type Completion cycle begins.

Registers are not needed to save the values of busA and bus B because IRWr is zero so busA and bus B will not change after the Register Fetch cycle.

Similarly, a register is not needed to save the ALU output at the end of the R-type execution cycle because:

1. We already established that Bus A and busB will not change the values.
2. And with control signals ALUSelA, ALUSelB, and ALUOp remains constant.
3. ALU output will not change its value during the R-type completion stage.

\[ +3 = 43 \text{ min.} \ (Y:23) \]
In theory, you ONLY need a register to hold a signal value if:

1. The signal is computed in one clock cycle and used in another.
2. AND the inputs to the functional block that computes this signal can change before this signal is written into a state element.

In other words, you do not need a register even if Cond 1 is true as long as Cond 2 is not true. That is as long as the input to the functional element that computes the signal does not change, we do not need a register to save its output. This is the case in our example here.

However, in practice, if you do not use a register to save a signal’s value when Cond 1 is true, you will introduce a multiple cycle delay path into your design.

By definition, a Multiple Cycle Delay path is a COMBINATIONAL logic delay path that takes multiple cycles to propagate from one storage element to the next storage element.

Let me show you what I mean by this.

\[ +2 = 45 \text{ min. (Y:25)} \]
Pros and Cons of a Multiple Cycle Delay Path

- A 3-cycle path example:
  - IR (storage) → Reg File Read → ALU → Reg File Write (storage)

- Advantages:
  - Register savings
  - We can share time among cycles:
    - If ALU takes longer than one cycle, still “a OK” as long as the entire path takes less than 3 cycles to finish

For example, the datapath activities during the Register Fetch (Reg File), R-type Execution (ALU), and R-type Completion (Feedback) is a 3-cycle delay path.

The 2 storage elements in this path are the Instruction Register and the Register File’s WRITE port. Remember the Register File’s Read port acts like a combinational logic path.

So in three cycles, we need to propagate the signals from the Instruction Register, through the Register File’s Read Port, through the ALU, and finally into the Register File’s WR port.

So what is the advantages of a multiple cycle datapath?

Well the obvious one is register savings. Here we save 32 register bits at Bus A, 32 register bits at Bus B, and another 32 register bits at ALU output. A total saving of 96 register bits.

But register bits are cheap. Another advantage of a multiple cycle delay path is that it allows us to share time among different cycles.

For example here, if we have a slow ALU that has a delay longer than one clock cycle, we will still meet the timing requirement AS LONG AS the total delay is less than three cycles.

\[ +2 = 47 \text{ min. (Y:27)} \]
The main disadvantage of having multiple cycle delay paths in your design is that static timing analyzers will report this as a timing violation.

A static timing analyzer is a CAD tool that looks at every combinational logic delay path between any two storage elements and report the delay of that path.

Consequently, the static timing analyzer will report this as a violation because it will think, and I must say correctly, that this path will take three times the cycle time to finish.

It will be OK if there is ONLY one multiple cycle delay path in your design because all you will do is look at the timing analyzer output and say, YEAH, I know this takes 3 times the cycle time to finish but it is OK and ignore the timing analyzer's warning.

However, if you have hundreds of them in your design, then you have to look at the timing analyzer's violation report one by one and decide whether to ignore it in a case by case basis.

Needless to say, this is a very tedious and error prone task and you may end up ignoring some real timing violations which you think are legitimate multiple cycle delay path.

Therefore try to avoid having multiple cycle path in your design as much as possible by using a register to save a signal's value whenever you generate a signal in one cycle and not use it until another cycle later.

For example here, I will put in registers to save the values of BusA and BusB between the Register Fetch and R-type Execution cycle and a register here to save the values of the ALU output between the R-type Execution and R-type Completion cycle.

Due to the space I have on each slide and I also like to try to keep this datapath as similar to the one in your text book as possible, I will not put in these registers in the lecture slides.

However, I do recommend you to avoid multiple cycle delay path in your design work.

+3 = 50 min. (Y:30)
Let’s go back to the end of the Register Fetch slash Instruction Decode Cycle.
Assume the result of the Instruction Decode indicates we have a OR immediate instruction, what do we do then? Well, we go to the OR immediate execution cycle.

+1 = 56 min. (Y:36)
The first operand of OR immediate comes from register Rs. It is already on busA so we just set ALUSelA to 1.

The second operand, on the other hand, does NOT come from Rt. It comes from the Zero Extended (ExtOp = 0) version of the immediate field (ALUSelB = 11).

Once we have the operands, all we have to do is to ask the ALU to OR (ALUop) them together and the ALU output will have the correct result at the end of this cycle.

Notice that I have set RegDst to zero so the Rt field of the instruction word will be stable at Register File’s Rw address port before the next cycle.

What do we do in the next cycle?

+2 = 58 min. (Y:38)
Well we do a register write (RegWt=1).

Once again, I have set up the register write address Rw in advance (RegDst = 0) during the previous cycle so I can guarantee Rw is stable when I assert RegWt in this cycle.

Also, remember we have a multiple cycle delay path from the Instruction Register to the Register File Write Port.

Therefore, IRWr must be 0 and ALUSelA & B, and ALUOp must remain the same as the previous cycle in order to guarantee ALU output to be stable during register write.

\[ +1 = 59 \text{ min. (Y:39)} \]
Let’s go back to the end of the Register Fetch slash Instruction Decode Cycle.
Assume the result of the Instruction Decode indicates we have a memory access instruction, that is we either have a load or store.

The next cycle we need to get into is the Memory Address Calculation cycle.

+1 = 60 min. (Y:40)
How do we calculate the memory address?

Simple, we have to add the contents of register Rs (busA) to the Sign Extended (ExtOp=1) version of the Immediate field (ALUSelB = 11).

With the ALUOp set to add, the memory address will be valid at the ALU output by the end of this cycle.

Let’s say we do have a store instruction and see what happens next.

\[ +1 = 61 \text{ min.} \ (Y:41) \]
Well, the address is already set up at the Memory's write address port. The data is also already available on the Memory's data port via busB. Therefore, all we have to do is to set MemWr to 1.

Notice that it is very important that we keep ALUSelA, ALUSelB, and ALUOp the same as the previous cycle, the Memory Address calculation cycle. Otherwise, if any of these control signals changes during Memory Write, the address will also change because we do not have a register to save the ALU output.

Any changes in the address during this cycle with MemWr = 1 will have catastrophic result. We will end up destroying data stored in memory by writing to the wrong address location.

+2 = 63 min. (Y:43)
If after the Memory Address calculation cycle, we realize we have a load.
We then enter the Load Memory Access cycle.
All we have to do is set the control signal IorD to 1 then after the memory read access delay, the data we want will be available at the output of the Ideal Memory (Dout).
Once again, we need to set RegDst to zero in this cycle so Rt will be stabilized at the Register file's write address port (Rw) before next cycle.

+2 = 45 min. (Y:45)
Because in this next cycle, the Write Back cycle, we will write the data from memory (MemtoReg = 1) into the register specified by the Rt field of the instruction.

+1 = 66 min. (Y:46)
Putting it all together, here it is: the multiple cycle datapath we set out to build.

+1 = 47 min. (Y:47)
Let me summarize what we learned today.

First we look at the single cycle processor we built and pointed out its two disadvantages:

1. First of all, it has a long cycle time.
2. And may be more importantly, this long cycle time is too long for all instructions except the Load.

I then show you how to design a multiple cycle processor by:

1. Divide the instructions into smaller steps.
2. Then execute each step (instead of the entire instruction) in one cycle.

Do NOT confuse Multiple Cycle Processor with Multiple Cycle Delay Path:

1. Multiple Cycle Processor executes each instruction in multiple clock cycles.
2. Multiple Cycle Delay Path: a combinational logic path between two storage elements that takes more than one clock cycle to complete.

It is possible (desirable) to build a MC Processor without MCDP:

1. Use a register to save a signal’s value whenever a signal is generated in one clock cycle and used in another cycle later.

---

Summary

- Disadvantages of the Single Cycle Processor
  - Long cycle time
  - Cycle time is too long for all instructions except the Load

- Multiple Cycle Processor:
  - Divide the instructions into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle

- Do NOT confuse Multiple Cycle Processor with Multiple Cycle Delay Path:
  - Multiple Cycle Processor executes each instruction in multiple clock cycles
  - Multiple Cycle Delay Path: a combinational logic path between two storage elements that takes more than one clock cycle to complete

- It is possible (desirable) to build a MC Processor without MCDP:
  - Use a register to save a signal’s value whenever a signal is generated in one clock cycle and used in another cycle later.
Well, we pretty much concentrated on the multiple cycle datapath today. But if you think about it, by summarizing all the control signals in circles along the way, we have pretty much specified the control in a state diagram.

All instructions start out at the Instruction Fetch cycle and continue to the Instruction Decode slash Register Fetch cycle. Once the instruction is decoded, we will either go to the Branch Complete cycle to complete the branch or go to one of the following:

(1) R-type execution or OR immediate execution for R-type or Or immediate instructions.
(2) Or we will go to the memory address calculation cycle for load and store instruction. The rest is pretty straight forward.

\[ +5 = 75 \text{ min. (Y:55)} \]
Where to get more information?

° Next two lectures:
   - Multiple Cycle Controller: Appendix C of your text book.
   - Microprogramming: Section 5.5 of your text book.


° Homework. See the website.