Designing a Single Cycle Datapath

Processor Design: How to Implement MIPS

Simplicity favors regularity

Start: X:40
Before we go any further, let’s step back for a second and take a look at the big picture. All computer consist of five components: (1) Input and (2) output devices. (3) The Memory System. And the (4) Control and (5) Datapath of the Processor.

Today’s lecture covers the datapath design.

In the next lecture, I will show you how to design the processor’s control unit.

+1 = 5 min. (X:45)
This slide shows how the next two lectures fit into the overall performance picture.
Recall from one of your earlier lectures that the performance of a machine is determined by 3 factors: (a) Instruction count, (b) Clock cycle time, and (c) Clock cycles per instruction.

Instruction count is controlled by the Instruction Set Architecture and the compiler design so the computer engineer has very little control over it (Instruction Count).

What you as a computer engineer can control, while you are designing a processor, are the Clock Cycle Time and Instruction Count per cycle.

More specifically, in the next two lectures, you will be designing a single cycle processor which by definition takes one clock cycle to execute every instruction.

The disadvantage of this single cycle processor design is that it has a long cycle time.

$+2 = 7 \text{ min.} \ (X:47)$
One of the most important things you need to know before you start designing a processor is how the instructions look like.

Or in more technical terms, you need to know the instruction format. One good thing about the MIPS instruction set is that it is very simple.

First of all, all MIPS instructions are 32 bits long and there are only three instruction formats: (a) R-type, (b) I-type, and (c) J-type.

The different fields of the R-type instructions are:
(a) OP specifies the operation of the instruction.
(b) Rs, Rt, and Rd are the source and destination register specifiers.
(c) Shamt specifies the amount you need to shift for the shift instructions.
(d) Funct selects the variant of the operation specified in the “op” field.

For the I-type instruction, bits 0 to 15 are used as an immediate field. I will show you how this immediate field is used differently by different instructions.

Finally, for the J-type instruction, bits 0 to 25 become the target address of the jump.

+3 = 10 min. (X:50)
In today’s lecture, I will show you how to implement the following subset of MIPS instructions: add, subtract, or immediate, load, store, branch, and the jump instruction.

The Add and Subtract instructions use the R format. The Op together with the Func fields together specified all the different kinds of add and subtract instructions.

Rs and Rt specifies the source registers. And the Rd field specifies the destination register.

The Or immediate instruction uses the I format. It only uses one source register, Rs. The other operand comes from the immediate field. The Rt field is used to specified the destination register.

Both the load and store instructions use the I format and both add the Rs and the immediate filed together to form the memory address.

The difference is that the load instruction will load the data from memory into Rt while the store instruction will store the data in Rt into the memory.

The branch on equal instruction also uses the I format. Here Rs and Rt are used to specified the registers we need to compare.

If these two registers are equal, we will branch to a location specified by the immediate field.

Finally, the jump instruction uses the J format and always causes the program to jump to a memory location specified in the address field.

I know I went over this rather quickly and you may have missed something. But don’t worry, this is just an overview. You will keep seeing these (point to the format) all day today.

+3 = 13 min. (X:53)
One thing you may notice from our last slide is that almost all instructions, except Jump, require reading some registers, do some computation, and then do something else.

Therefore our datapath will look something like this.

For example, if we have an add instruction (points to the output of Instruction Memory), we will read the registers from the register file (Ra, Rb and then busA and busB).

Add the two numbers together (ALU) and then write the result back to the register file.

On the other hand, if we have a load instruction, we will first use the ALU to calculate the memory address.

Once the address is ready, we will use it to access the Data Memory.

And once the data is available on Data Memory’s output bus, we will write the data to the register file. Well, this is simple enough.

But if it is this simple, you probably won’t need to take this class.

So in today’s lecture, I will show you how to turn this abstract datapath into a real datapath by making it slightly (JUST slightly) more complicated so it can do real work for you.

But before we do that, let’s do a quick review of the clocking methodology

\[ +3 = 16 \] (X:56)
Remember, we will be using a clocking methodology where all storage elements are clocked by the same clock edge. Consequently, our cycle time will be the sum of:

(a) The Clock-to-Q (or latch propagation) time of the input registers.
(b) The longest delay path through the combinational logic block.
(c) The set up time of the output register.
(d) And finally the clock skew.
In order to avoid hold time violation, you have to make sure this inequality is fulfilled.

\[ +2 = 18 \text{ min.} \] (X:58)
Now with the clocking methodology back in your mind, we can think about how the critical path of our “abstract” datapath may look like.

One thing to keep in mind about the Register File and Ideal Memory (points to both Instruction and Data) is that the Clock input is a factor ONLY during the write operation.

For read operation, the CLK input is not a factor. The register file and the ideal memory behave as if they are combinational logic.

That is you apply an address to the input, then after certain delay, which we called access time, the output is valid.

We will come back to these points (point to the “behave” bullets) later in this lecture.

But for now, let’s look at this “abstract” datapath’s critical path which occurs when the datapath tries to execute the Load instruction.

The time it takes to execute the load instruction are the sum of:

(a) The PC’s clock-to-Q time.
(b) The instruction memory access time.
(c) The time it takes to read the register file.
(d) The ALU delay in calculating the Data Memory Address.
(e) The time it takes to read the Data Memory.
(f) And finally, the setup time for the register file and clock skew.

\[ +3 = 21 \text{ (Y:01)} \]
So let’s design a processor. How and where do we start?

Well, the best place to start is the processor’s instruction set architecture. After all, the goal of your design is to execute the instructions in the instruction set correctly.

What you need to do is to describe each instruction’s operation in register transfer language. By looking at the Register Transfer Language description of the instruction, you can figure out the datapath components you need and how to connect these components together.

As I will show you, each datapath component will have its own set of control signals. And the last step of the processor design task is to design the control unit that generates the control signals for the datapath.

So what do we mean by Register Transfer Language?

\[ +2 = 27 \text{ min. (Y:07)} \]
Here is an example. In terms of Register Transfer Language, this is what the Add instruction need to do.

First, you need to fetch the instruction from memory.
Then you perform the actual add operation.
And finally, you need to update the program counter to point to the next instruction.

+1 = 28 min. (Y:08)
What is RTL: The Load Instruction

\[ \text{l} \text{w} \ rt, rs, \text{imm16} \]

- \( \text{mem}[\text{PC}] \) Fetch the instruction from memory
- \( \text{Addr} \leftarrow R[rs] + \text{SignExt}(\text{imm16}) \) Calculate the memory address
- \( R[rt] \leftarrow \text{Mem}[\text{Addr}] \) Load the data into the register
- \( \text{PC} \leftarrow \text{PC} + 4 \) Calculate the next instruction’s address

Here is another example.
The load instruction also starts off by fetching the instruction from Instruction Memory.
Then you calculate the memory address, use the address to fetch the data from memory
(\( \text{Mem}(\text{Addr}) \)), and then load the data into the register.
Finally, you need to update the PC to point to the next sequential instruction.

\[ +1 = 29 \text{ min} \ (Y:09) \]
Based on the Register Transfer Language examples we have so far, we know we will need the following combinational logic elements.

We will need an adder to update the program counter.
A MUX to select the results.
And finally, an ALU to do various arithmetic and logic operation.

+1 = 30 min. (Y:10)
As far as storage elements are concerned, we will need a N-bit register that is similar to the D flip-flop I showed you in class.

The significant difference here is that the register will have a Write Enable input. That is the content of the register will NOT be updated if Write Enable is zero. The content is updated at the clock tick ONLY if the Write Enable signal is set to 1.

+1 = 31 min. (Y:11)
We will also need a register file that consists of 32 registers with two output busses (busA and busB) and one input bus.

The register specifiers Ra and Rb select the registers to put on busA and busB respectively. When Write Enable is 1, the register specifier Rw selects the register to be written via busW.

In our simplified version of the register file, the write operation will occur at the clock tick. Keep in mind that the clock input is a factor ONLY during the write operation.

During read operation, the register file behaves as a combinational logic block. That is if you put a valid value on Ra, then bus A will become valid after the register file’s access time.

Similarly if you put a valid value on Rb, bus B will become valid after the register file’s access time. In both cases (Ra and Rb), the clock input is not a factor.

\[ +2 = 33 \text{ min.} \]
We will also need a register file that consists of 32 32-bit registers with two output busses (busA and busB) and one input bus.

The register specifiers Ra and Rb select the registers to put on busA and busB respectively. When Write Enable is 1, the register specifier Rw selects the register to be written via busW.

In our simplified version of the register file, the write operation will occur at the clock tick. Keep in mind that the clock input is a factor ONLY during the write operation.

During read operation, the register file behaves as a combinational logic block. That is if you put a valid value on Ra, then bus A will become valid after the register file’s access time.

Similarly if you put a valid value on Rb, bus B will become valid after the register file’s access time. In both cases (Ra and Rb), the clock input is not a factor.

\[+2 = 33 \text{ min. (Y:13)}\]
The last storage element you will need for the datapath is the idealized memory to store your data and instructions. This idealized memory block has just one input bus (DataIn) and one output bus (DataOut). When Write Enable is 0, the address selects the memory word to put on the Data Out bus. When Write Enable is 1, the address selects the memory word to be written via the Data In bus at the next clock tick. Once again, the clock input is a factor ONLY during the write operation. During read operation, it behaves as a combinational logic block. That is if you put a valid value on the address lines, the output bus DataOut will become valid after the access time of the memory.

+2 = 35 min. (Y:15)
Now let’s take a look at the first major component of the datapath: the instruction fetch unit. The common RTL operations for all instructions are:

(a) Fetch the instruction using the Program Counter (PC) at the beginning of an instruction’s execution (PC -> Instruction Memory -> Instruction Word).

(b) Then at the end of the instruction’s execution, you need to update the Program Counter (PC -> Next Address Logic -> PC).

More specifically, you need to increment the PC by 4 if you are executing sequential code. For Branch and Jump instructions, you need to update the program counter to “something else” other than plus 4.

I will show you what is inside this Next Address Logic block when we talked about the Branch and Jump instructions.

For now, let’s focus our attention to the Add and Subtract instructions.

+2 = 37 min. (Y:17)
The Add instruction is a R-type instruction.
The Instruction Fetch Unit I just showed you will take care of the instruction fetch (mem[PC]) and PC update (PC <- PC + 4).
The thing we need to take care of now is the actual operation: add the contents of the registers specified by the Rs and the Rt fields (Rs and Rt of the format diagram).
And then write the results to the register specified by the Rd field.

+1 = 38 min. (Y:18)
RTL: The Subtract Instruction

```
sub rd, rs, rt
```

- mem[PC]  
  Fetch the instruction from memory

- R[rd] <- R[rs] - R[rt]  
  The actual operation

- PC <- PC + 4  
  Calculate the next instruction’s address

The Subtract instruction is also a R-type instruction.
Here we need to subtract the the contents of the register specified by Rt from the contents of the register specified by the Rs field (Rs and Rt of the format diagram).
And then write the results back to the register specified by the Rd field.

\[ +1 = 39 \text{ min. (Y:19)} \]
And here is the datapath that can do the trick.
First of all, we connect the register file’s Ra, Rb, and Rw input to the Rd, Rs, and Rt fields of the instruction bus (points to the format diagram).
Then we need to connect busA and busB of the register file to the ALU.
Finally, we need to connect the output of the ALU to the input bus of the register file. Conceptually, this is how it works.
The instruction bus coming out of the Instruction memory will set the Ra and Rb to the register specifiers Rs and Rt.
This causes the register file to put the value of register Rs onto busA and the value of register Rt onto busB, respectively.
But setting the ALUctr appropriately, the ALU will perform either the Add and Subtract for us.
The result is then fed back to the register file where the register specifier Rw should already be set to the instruction bus’s Rd field.
Since the control, which we will design in our next lecture, should have already set the RegWr signal to 1, the result will be written back to the register file at the next clock tick (points to the Clk input).

+3 = 42 min. (Y:22)
Let’s take a more quantitative picture of what is happening.

At each clock tick, the Program Counter will present its latest value to the Instruction memory after Clk-to-Q time.

After a delay of the Instruction Memory Access time, the Opcode, Rd, Rs, Rt, and Function fields will become valid on the instruction bus.

Once we have the new instruction, that is the Add or Subtract instruction, on the instruction bus, two things happen in parallel.

First of all, the control unit will decode the Opcode and Func field and set the control signals ALUctr and RegWr accordingly. We will cover this in the next lecture.

While this is happening (points to Control Delay), we will also be reading the register file (Register File Access Time).

Once the data is valid on busA and busB, the ALU will perform the Add or Subtract operation based on the ALUctr signal.

Hopefully, the ALU is fast enough that it will finish the operation (ALU Delay) before the next clock tick.

At the next clock tick, the output of the ALU will be written into the register file because the RegWr signal will be equal to 1.

\[ +3 = 45 \text{ min. (Y:25)} \]
The or immediate is a I-type instruction. The immediate field of the instruction (Imm16 of the format diagram) is zero extended to 32 bits before it is operated with the other operand. The other operand is selected by the Rs field of the instruction. The destination register of this instruction will be selected by the Rt field.

+2 = 57 min. (Y:27)
Here is the datapath for the Or immediate instructions.

We cannot use the Rd field here (Rw) because in this instruction format, we don’t have a Rd field. The Rd field in the R-type is used here as part of the immediate field.

For this instruction type, Rw input of the register file, that is the address of the register to be written, comes from the Rt field of the instruction.

Recalled from earlier slide that for R-type instruction, the Rw comes from the Rd field. That’s why we need a MUX here to put Rd onto Rw for R-type instructions and to put Rt onto Rw for the I-type instruction.

Since the second operation of this instruction will be the immediate field zero extended to 32 bits, we also need a MUX here to block off bus B from the register file.

Since bus B is blocked off by the MUX, the value on bus B is don’t care. Therefore we do not have to worry about what ends up on the register file’s Rb register specifier.

To keep things simple, we may just as well keep it the same as the R-type instruction and put the Rt field here.

So to summarize, this is how this datapath works. With Rs on Register File’s Ra input, bus A will get the value of Rs as the first ALU operand.

The second operand will come from the immediate field of the instruction.

Once the ALU complete the OR operation, the result will be written into the register specified by the instruction’s Rt field.

+3 = 50 min. (Y:30)
Like the OR immediate instruction I just showed you, the load instruction also uses the I format (point to the format diagram).

But unlike the OR immediate instruction, the immediate field (Imm16 of the format diagram) is sign extended instead of zero extended.

That is we will duplicate the most significant bit of 16 times to the left to form a 32-bit value. This sign extended value (SignExt) is then added to the register selected by the Rs field of the instruction to form the memory address.

The memory address is then used to load the value into the register specified by the Rt field of the instruction (Rt of the format diagram).

+2 = 57 min. (Y:37)
Once again we cannot use the instruction’s Rd field for the Register File’s Rw input because load is a I-type instruction and there is no such thing as the Rd field in the I format.

So instead of Rd, the Rt field is used to specify the destination register through this two to one multiplexor.

The first operand of the ALU comes from busA of the register file which contains the value of Register Rs (points to the Ra input of the register file).

The second operand, on the other hand, comes from the immediate field of the instruction. Instead of using the Zero Extender I used in datapath for the or immediate datapath, I have to use a more general purpose Extender that can do both Sign Extend and Zero Extend.

The ALU then adds these two operands together to form the memory address. Consequently, the output of the ALU has to go to two places:

(a) First the address input of the data memory.

(b) And secondly, also to the input of this two-to-one multiplexer. The other input of this multiplexer comes from the output of the data memory so we can place the output of the data memory onto the register file’s input bus for the load instruction.

For Add, Subtract, and the Or immediate instructions, the output of the ALU will be selected to be placed on the input bus of the register file.

In either case, the control signal RegWr should be asserted so the register file will be written at the end of the cycle.

\[+3 = 60 \text{ min. (Y:40)}\]
Just like the load instruction:
(a) The store instruction also uses the I format.
(b) And the store instruction also forms the memory address by adding the contents of the register selected by the Rs field to the sign extended immediate field.

However, unlike the load instruction, which gets data from memory and put the data into the register file, the store instruction:
(a) Get the register selected by the Rt field of the instruction (R[rt]).
(b) And then write this register into the data memory.

+2 = 62 min. (Y:42)
And here is the datapath for the store instruction. The Register File, the ALU, and the Extender are the same as the datapath for the load instruction because the memory address has to be calculated the exact same way:

(a) Put the register selected by Rs onto bus A and sign extend the 16 bit immediate field.

(b) Then make the ALU (ALUctr) adds these two (busA and output of Extender) together.

The new thing we added here is busB extension (DataIn).

More specifically, in order to send the register selected by the Rt field (Rb of the register file) to data memory, we need to connect bus B to the data memory’s Data In bus.

Finally, the store instruction is the first instruction we encountered that does not do any register write at the end.

Therefore the control unit must make sure RegWr is zero for this instruction.

\[ +2 = 64 \text{ min.} \ (Y:44) \]
How does the branch on equal instruction work?

Well it calculates the branch condition by subtracting the register selected by the Rt field from the register selected by the Rs field.

If the result of the subtraction is zero, then these two registers are equal and we take a branch. Otherwise, we keep going down the sequential path (PC <- PC + 4).
The datapath for calculating the branch condition is rather simple. All we have to do is feed the Rs and Rt fields of the instruction into the Ra and Rb inputs of the register file. Bus A will then contain the value from the register selected by Rs. And bus B will contain the value from the register selected by Rt. The next thing to do is to ask the ALU to perform a subtract operation and feed the output Zero to the next address logic. How does the next address logic block look like? Well, before I show you that, let’s take a look at the binary arithmetics behind the program counter (PC).

\[ +2 = 67 \text{ min. (Y:47)} \]
In theory, the Program Counter (PC) is a 32-bit byte address into the Instruction memory. The Program Counter is incremented by four after each sequential instruction.

When a branch is taken, we need to sign extend the 16-bit immediate field, multiply this sign extended value by four, and add it to the sequential instruction address (PC + 4).

Why does this magic number “4” always come up? Well, the reason is that the 32-bit PC is a byte address and all MIPS instructions are four bytes (32 bits) long.

In other words:
- The 2 least significant bits of the 32-bit PC are always zeros
- There is no reason to have hardware to keep the 2 least significant bits

In practice, we can simplify the hardware by using a 30-bit PC<31:2>:
- Sequential operation: PC<31:2> = PC<31:2> + 1
- Branch operation: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
- In either case: Instruction Memory Address = PC<31:2> concat “00”

In theory, the Program Counter (PC) is a 32-bit byte address into the Instruction memory. The Program Counter is incremented by four after each sequential instruction.

When a branch is taken, we need to sign extend the 16-bit immediate field, multiply this sign extended value by four, and add it to the sequential instruction address (PC + 4).

Why does this magic number “4” always come up? Well, the reason is that the 32-bit PC is a byte address and all MIPS instructions are four bytes (32 bits) long.

In other words, if we keep a 32-bit Program Counter, then the two least significant bits of the Program Counter will always be zeros.

And if these two bits are always zeros, there is no reason to have hardware to keep them.

So in practice, we will simply the hardware by using a 30-bit program counter.

That is, we will build a Program Counter that only keeps track of the upper 30 bits (<31:2>) of the instruction address because we know the two least significant bits will always be zeros.

Then instead of always increase the Program Counter by four for sequential operation, we only have to increase it by 1.

And for branch operation, we don’t need to multiply the sign extended immediate field by four before adding to the sequential PC (PC + 1).

And when we apply the program counter to the address of the instruction memory, we need to attach two zeros to its least significant bits.

+3 = 70 min. (Y:50)
Next Address Logic: Expensive and Fast Solution

- Using a 30-bit PC:
  - Sequential operation: PC<31:2> = PC<31:2> + 1
  - Branch operation: PC<31:2> = PC<31:2> + 1 + SignExt[Imm16]
  - In either case: Instruction Memory Address = PC<31:2> concat "00"

So let’s see how we can put all these theories (point to the equations) into practice.

The PC plus one is implemented by this first adder here.

For branch operation, we need to sign extend the immediate field of the instruction and then add it to the output of the first adder to implement this equation (PC + 1 + SignExt[Imm16]).

For sequential operation, the output of the first adder is selected by the two-to-one mux so it will be saved into the PC register at the next clock tick.

For a taken branch, that is we have a branch_on_equal and the condition Zero is true, the output of the second adder is selected.

In all cases, the 30 bit Program Counter is used as instruction address bit 31 to bit 2.

The two least significant bits of the instruction address will always be zeroes.

One question you may want to ask is: Do we really need an adder just to add “1”?

Well may be not.

+2 = 72 min. (Y:52)
One way to simplify the implementation is to use the CarryIn input of the adder to implement the $PC_{31:2} = PC_{31:2} + 1$ operation.

Then we can put a MUX in front of the adder to add the branch offset if the branch is taken.

If the branch is not taken, we simply set the 2nd output of the ALU to zeros so we only add one through the CarryIn input.

Why is this implementation slow?

Well because we cannot start the address add until the Zero input is valid.

And when will the Zero input become valid? Not until we have performed a 32-bit subtract in the main datapath.

But does it matter that this is slow in the overall scheme of things?

Well, probably not in this single cycle implementation.

The critical path of this single cycle implementation will be the load instruction’s memory access so the extra time it takes to calculate the PC can be hidden behind the critical path.

$+3 = 75 \text{ min (Y:55)}$
Finally, let's take a look at the jump instruction which uses the J format.

The effect of the jump instruction is to change the lower 26 bits of the Program Counter to the value specified in the address field of the instruction.

\[ +1 = 76 \text{ min. (Y:46)} \]
Well this (points to the equation) is easy to implement.
All we have to do is grab the four most significant bits of the PC and put them right next to the 26
bits target, and we will have the next PC for the jump (point to the feedback path).
If we are running Powerview, what we will do now is to create a symbol called Instruction Fetch
Unit.
The output of this symbol is the 32-bit instruction word.
The input to the Instruction Fetch Unit are two control signals, Branch and Jump, and one
conditional input Zero from the datapath.
Using this new symbol, we can complete our single cycle datapath.

+2 = 78 min. (Y:58)
So here is the single cycle datapath we just built.

If you push into the Instruction Fetch Unit, you will see the last slide showing the PC, the next address logic, and the Instruction Memory.

Here I have shown how we can get the Rt, Rs, Rd, and Imm16 fields out of the 32-bit instruction word.

The Rt, Rs, and Rd fields will go to the register file as register specifiers while the Imm16 field will go to the Extender where it is either Zero and Sign extended to 32 bits.

The signals ExtOp, ALUSrc, ALUctr, MemWr, MemtoReg, RegDst, RegWr, Branch, and Jump are control signals.

And I will show you how to generate them in the next class..

+2 = 80 min. (Z:00)
So that's all for today. See you guys Thursday.