ECE4680
Computer Organization and Architecture

Designing Single Cycle Control

How to design a controller to produce signals to control the datapath
Recap: The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. The three instruction formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td></td>
<td>op [31:26] rs [25:21] rt [20:16] rd [15:11] shamt [10:5] funct [4:0]</td>
</tr>
<tr>
<td>I-type</td>
<td></td>
<td>op [31:26] rs [25:21] rt [20:16] immediate [15:0]</td>
</tr>
<tr>
<td>J-type</td>
<td></td>
<td>op [31:26] rs [25:21] rt [20:16] target address [25:0]</td>
</tr>
</tbody>
</table>

- The different fields are:
  - op: operation of the instruction
  - rs, rt, rd: the source and destination registers specifier
  - shamt: shift amount
  - funct: selects the variant of the operation in the “op” field
  - address / immediate: address offset or immediate value
  - target address: target address of the jump instruction

In our last lecture, I show you how to implement the datapath for a subset of the MIPS instruction set. Here is a quick review of the MIPS 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.

+2 = 2 min. (X:42)
Recap: The MIPS Subset

° ADD and subtract
  • add rd, rs, rt
  • sub rd, rs, rt

° OR Imm:
  • ori rt, rs, imm16

° LOAD and STORE
  • lw rt, rs, imm16
  • sw rt, rs, imm16

° BRANCH:
  • beq rs, rt, imm16

° JUMP:
  • j target

+3 = 5min. (X:45)
Recap: A Single Cycle Datapath

- We have everything except control signals (underline)
- Today’s lecture will show you how to generate the control signals

The result of the last lecture is this single-cycle datapath.

+1 = 6 min. (X:46)
The Big Picture: Where are We Now?

° The Five Classic Components of a Computer

° Today’s Topic: Designing the Control for the Single Cycle Datapath

So where are in in the overall scheme of things.  
Well, we just finished designing the processor’s datapath.  
Now I am going to show you how to design the control for the datapath.

+1 = 7 min. (X:47)
OK, let’s get on with today’s lecture by looking at the simple add instruction.

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. More specifically:
(a) You add the contents of the register specified by the Rs and Rt fields of the instruction.
(b) Then you write the results to the register specified by the Rd field.
And finally, you need to update the program counter to point to the next instruction.
Now, let’s take a detail look at the datapath during various phase of this instruction.

+2 = 10 min. (X:50)
First let’s look at the Instruction Fetch Unit where everything begins. Every instruction begins at the clock tick. The clock tick in this case is the high to low transition of the Clk (points to the “bubble” of PC).

What happens right after the clock tick?

After Clk-to-Q delay, the PC gets the value that points to the Add instruction and fetch the add instruction from the memory but sending the address to the Ideal Instruction memory.

Notice that since this is the beginning of the instruction, Control signals Branch and Jump will still have the old values from the previous instruction.

At the beginning of ALL instructions execution, the instruction unit behaves the same way as shown here and we won’t repeat this picture for every instruction.

+2 = 12 min. (X:52)
This picture shows the activities at the main datapath during the execution of the Add or Subtract instructions.

The active parts of the datapath are shown in different color as well as thicker lines.

First of all, the Rs and Rt of the instructions are fed to the Ra and Rb address ports of the register file and cause the contents of registers specified by the Rs and Rt fields to be placed on busA and busB, respectively.

With the ALUctr signals set to either Add or Subtract, the ALU will perform the proper operation and with MemtoReg set to 0, the ALU output will be placed onto busW.

The control we are going to design will also set RegWr to 1 so that the result will be written to the register file at the end of the cycle.

Notice that ExtOp is don’t care because the Extender in this case can either do a SignExt or ZeroExt. We DON’T care because ALUSrc will be equal to 0--we are using busB.

The other control signals we need to worry about are:

(a) MemWr has to be set to zero because we do not want to write the memory.

(b) And Branch and Jump, we have to set to zero. Let me show you why.

+3 = 15 min. (X:55)
This picture shows the control signals setting for the Instruction Fetch Unit at the end of the Add or Subtract instruction.

Both the Branch and Jump signals are set to 0.

Consequently, the output of the first adder, which implements PC plus 1, is selected through the two 2-to-1 mux and got placed into the input of the Program Counter register.

The Program Counter is updated to this new value at the next clock tick.

Notice that the Program Counter is updated at every cycle. Therefore it does not have a Write Enable signal to control the write.

Also, this picture is the same for or all instructions other than Branch and Jump.

Therefore I will only show this picture again for the Branch and Jump instructions and will not repeat this for all other instructions.

+2 = 17 min. (X:57)
Now let’s look at the control signals setting for the Or immediate instruction.

The OR immediate instruction OR the content of the register specified by the Rs field to the Zero Extended Immediate field and write the result to the register specified in Rt.

This is how it works in the datapath. The Rs field is fed to the Ra address port to cause the contents of register Rs to be placed on busA.

The other operand for the ALU will come from the immediate field. In order to do this, the controller need to set ExtOp to 0 to instruct the extender to perform a Zero Extend operation.

Furthermore, ALUSrc must set to 1 such that the MUX will block off bus B from the register file and send the zero extended version of the immediate field to the ALU.

Of course, the ALUctr has to be set to OR so the ALU can perform an OR operation.

The rest of the control signals (MemWr, MemtoReg, Branch, and Jump) are the same as the Add and Subtract instructions.

One big difference is the RegDst signal. In this case, the destination register is specified by the instruction’s Rt field, NOT the Rd field because we do not have a Rd field here.

Consequently, RegDst must be set to 0 to place Rt onto the Register File’s Rw address port.

Finally, in order to accomplish the register write, RegWr must be set to 1.

\[+3 = 20 \text{ min. (X:60)}\]
Let's continue our lecture with the load instruction. What does the load instruction do?

It first adds the contents of the register specified by the Rs field to the Sign Extended version of the Immediate field to form the memory address.

Then it uses this memory address to access the memory and write the data back to the register specified by the Rt field of the instruction.

Here is how the datapath works: first the Rs field is fed to the Register File’s Ra address port to place the register onto bus A.

Then the ExtOp signal is set to 1 so that the immediate field is Sign Extended and we place this value (output of Extender) onto the ALU input by setting ALUsrc to 1.

The ALU then add (ALUctr = add) the two together to form the memory address which is then placed onto the Data Memory’s address port.

In order to place the Data Memory’s output bus onto the Register File’s input bus (busW), the control needs to set MemtoReg to 1.

Similar to the OR immediate instruction I showed you earlier, the destination register here is specified by the Rt field. Therefore RegDst must be set to 0.

Finally, RegWr must be set to 1 to complete the register write operation.

Well, it should be obvious to you guys by now that we need to set Branch and Jump to 0 to make sure the Instruction Fetch Unit update the Program Counter correctly.

+3 = 28 min. (Y:08)
The store instruction performs the inverse function of the load. Instead of loading data from memory, the store instruction sends the contents of register specified by Rt to data memory.

Similar to the load instruction, the store instruction needs to read the contents of register Rs (points to Ra port) and add it to the sign extended version of the immediate filed (Imm16, ExtOp = 1, ALUSrc = 1) to form the data memory address (ALUctr = add).

However unlike the Load instruction where busB is not used, the store instruction will use busB to send the data to the Data memory.

Consequently, the Rt field of the instruction has to be fed to the Rb port of the register file.

In order to write the Data Memory properly, the MemWr signal has to be set to 1.

Notice that the store instruction does not update the register file. Therefore, RegWr must be set to zero and consequently control signals RegDst and MemtoReg are don’t cares.

And once again we need to set the control signals Branch and Jump to zero to ensure proper Program Counter updating.

Well, by now, you are probably tired of these boring stuff where Branch and Jump are zero so let’s look at something different--the branch instruction.

\[ +3 = 31 \text{ min. (Y:11)} \]
So how does the branch instruction work?

As far as the main datapath is concerned, it needs to calculate the branch condition. That is, it subtracts the register specified in the Rt field from the register specified in the Rs field and sets the condition Zero accordingly.

In order to place the register values on busA and busB, we need to feed the Rs and Rt fields of the instruction to the Ra and Rb ports of the register file and set ALUSrc to 0.

Then we have to instruction the ALU to perform the subtract (ALUctr = sub) operation and set the Zero bit accordingly.

The Zero bit is sent to the Instruction Fetch Unit. I will show you the internal of the Instruction Fetch Unit in a second.

But before we leave this slide, I want you to notice that ExtOp, MemtoReg, and RegDst are don’t cares but RegWr and MemWr have to be ZERO to prevent any write to occur.

And finally, the controller needs to set the Branch signal to 1 so the Instruction Fetch Unit knows what to do. So now let’s take a look at the Instruction Fetch Unit.

+2 = 33 min. (Y:13)
Let’s look at the interesting case where the branch condition Zero is true (Zero = 1). Well, if Zero is not asserted, we will have our boring case where PC + 1 is selected. Anyway, with Branch = 1 and Zero = 1, the output of the second adder will be selected. That is, we will add the sequential address, that is output of the first adder, to the sign extended version of the immediate field, to form the branch target address (output of 2nd adder).

With the control signal Jump set to zero, this branch target address will be written into the Program Counter register (PC) at the end of the clock cycle.

\[ +2 = 35 \text{ min. (Y:15)} \]
The control signals setting in the main datapath for the Jump instruction is pretty boring because in most cases, we DON'T CARE.

More specifically, control signals ExtOp, ALUSrc, ALUctr are all don't cares because the ALU is not used at all for the Jump instruction.

Control signals MemtoReg and RegDst are don't are because Jump does not write the register file. That is the reason why we still need to set RegWr to zero.

Furthermore, we also need to set MemWr to zero to avoid Data Memory write.

Finally, the control signal Branch is set to zero but Jump is set to 1.

\[ +2 = 37 \text{ min. (X:17)} \]
Inside the Instruction Fetch Unit, with Branch set to zero and Jump set to 1, we will not use the output of neither Adder.

What we will use is the concatenation of the four most significant bits of the current program counter and the twenty six bits of the target address.

With the control signal Jump set to 1, this value will be send to the Program Counter and get written into PC at the next clock tick (points to the Clk bubble).

\[ +2 = 39 \text{ min. (Y:19)} \]
Here is a table summarizing the control signals setting for the seven (add, sub, ...) instructions we have looked at.

Instead of showing you the exact bit values for the ALU control (ALUctr), I have used the symbolic values here.

The first two columns are unique in the sense that they are R-type instructions and in order to uniquely identify them, we need to look at BOTH the op field as well as the funct fields.

Ori, lw, sw, and branch on equal are I-type instructions and Jump is J-type. They all can be uniquely identified by looking at the opcode field alone.

Now let's take a more careful look at the first two columns. Notice that they are identical except the last row.

So we can combine these two rows here if we can “delay” the generation of ALUctr signals.

This lead us to something called “local decoding.”

+3 = 42 min. (Y:22)
The Concept of Local Decoding

<table>
<thead>
<tr>
<th>op</th>
<th>00 0000</th>
<th>00 1101</th>
<th>10 0011</th>
<th>10 1011</th>
<th>00 0100</th>
<th>00 0010</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>ori</td>
<td>lw</td>
<td>sw</td>
<td>beq</td>
<td>jump</td>
<td></td>
</tr>
<tr>
<td>RegDst</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Branch</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUop&lt;N:0&gt;</td>
<td>“R-type”</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>xxx</td>
</tr>
</tbody>
</table>

That is, instead of asking the Main Control to generates the ALUctr signals directly (see the diagram with the ALU), the main control will generate a set of signals called ALUop.

For all I and J type instructions, ALUop will tell the ALU Control exactly what the ALU needs to do (Add, Subtract, ...).

But whenever the Main Control sees a R-type instructions, it simply throws its hands up and say: “Wow, I don’t know what the ALU has to do but I know it is a R-type instruction” and let the Local Control Block, ALU Control to take care of the rest.

Notice that this save us one column from the table we had on the last slide. But let’s be honest, if one column is the ONLY thing we save, we probably will not do it.

But when you have to design for the entire MIPS instruction set, this column will used for ALL R-type instructions, which is more than just Add and Subtract I showed you here.

Another advantage of this table over the last one, besides being smaller, is that we can uniquely identify each column by looking at the Op field only.

Therefore, as I will show you later, the Main Control ONLY needs to look at the Opcode field.

How many bits do we need for ALUop?

+3 = 45 min. (Y:25)
Well the answer is 2 because we only need to represent 4 things: “R-type,” the Or operation, the Add operation, and the Subtract operation.

If you are implementing the entire MIPS instruction set, then ALUop has to be 3 bits wide because we will need to represent 5 things: R-type, Or, Add, Subtract, and AND.

Here I show you the bit assignment I made for the 3-bit ALUop.

With this bit assignment in mind, let’s figure out what the local control ALU Control has to do.

+1 = 26 min. (Y:26)
What this table and diagram implies is that if the ALU Control receives ALUop = 100, it has to decode the instruction’s "func" field to figure out what the ALU needs to do.

Based on the MIPS encoding in Appendix A of your text book, we know we have a Add instruction if the func field is 10 000.

If the func field is 10 0010, we know we have a subtract operation and so on.

Notice that the bit 5 and bit 4 of this field is the same for all these operations so as far as the ALU control is concerned, these bits are don’t care.

Now recall from your ALU homework, the ALUctr signals has the following meaning (point to the table): 000 means Add, 001 means subtract, ... etc.

Based on these three tables (point to the last row of the top table and then the two other tables) and the fact that bit 5 and bit 4 of the “func” field are don’t care, we can derive the following truth table for ALUctr.

\[ +2 = 48 \text{ min. (Y:28)} \]
That is, whenever ALUop is 000, we don’t care anything about the func field because we know we need the ALU to do an ADD operation (point to Add column).

Whenever the ALUop bit<2> is 0 and bit<0> is 1, we know we want the ALU to perform a Subtract regardless of what func field is.

Bit<1> is a don’t care because for our encoding here, ALUop<1> will never be equal to 1 whenever bit<0> is 1 and bit<2> is 0.

Similarly, whenever ALUop bit<2> is 0 and bit<1> is 1, we need the ALU to perform Or.

The tricky part occurs when the ALUOp bit<2> equals to 1. In that case, we have a R-type instruction and we need to look at the Func field.

In any case, once we have this Symbolic column, we can get this actual bit columns by referring to our ALU able on the last slide (use the last slide if time permit).

\[ +2 = 30 \text{ min.} \ (Y:30) \]
From the truth table we had before the break, we can derive the logic equation for ALUctr bit 2 but collecting all the rows that has ALUctr bit 2 equals to 1 and this table is the result.

Each row becomes a product term and we need to OR the product terms together.

Notice that the last row are identical except the bit<3> of the func fields. One is zero and the other is one. Together, they make bit<3> a don’t care term.

With all these don’t care terms, the logic equation is rather simple.

The first product term is: not ALUOp<2> and ALUOp<0>.

The second product term, after we making Func<3> a don’t care becomes ...

\[ \text{ALUctr}<2> = \neg \text{ALUOp}<2> \land \text{ALUOp}<0> + \text{ALUOp}<2> \land \neg \text{func}<2> \land \text{func}<1> \land \neg \text{func}<0> \]

From the truth table we had before the break, we can derive the logic equation for ALUctr bit 2 but collecting all the rows that has ALUct bit 2 equals to 1 and this table is the result.

Each row becomes a product term and we need to OR the product terms together.

Notice that the last row are identical except the bit<3> of the func fields. One is zero and the other is one. Together, they make bit<3> a don’t care term.

With all these don’t care terms, the logic equation is rather simple.

The first product term is: not ALUOp<2> and ALUOp<0>.

The second product term, after we making Func<3> a don’t care becomes ...

\[ +2 = 57 \text{ min. (Y:37)} \]
Here is the truth table when we collect all the rows where ALC\texttt{ctr} bit<1> equals to 1. Once again, we can simplify the table by noticing that the first two rows are different only at the ALU\texttt{op} bit<0> position. We can make ALU\texttt{op} bit<0> into a don’t care. Similarly, the last three rows can be combined to make Func bit<3> and bit<1> into don’t cares. Consequently, the logic equation for ALU\texttt{ctr} bit<1> becomes ...

\[ +2 = 59 \text{ min. (Y:39)} \]
Finally, after we gather all the rows where ALUctr bit 0 are 1’s, we have this truth table.

Well, we are out of luck here. I don’t see any simple way to simplify these product terms by just looking at them.

There may be some if you draw out the 7 dimension K map but I am not going to try it.

So I just write down the logic equations as it is.

\[ +2 = 61 \text{ min. (Y:41)} \]
With all the logic equations available, you should be able to implement this logic block without any problem.

+1 = 62 min. (Y:42)
Now that we have taken care of the Local Control (ALU Control), let's refocus our attention to the Main Controller.

The job of the Main Control is to look at the Opcode field of the instruction and generate these control signals for the datapath (RegDst, ... ExtOp) as well as the 3-bit ALUop field for the ALU Control.

Here, I have shown you the symbolic value of the ALUop field as well as the actual bit assignment. For example here (2nd column), the R-type ALUop is encoded as 100 and the Add operation (3rd column) is encoded as 000.

This is called a “Truth Table” because if you think about it, this is like having the truth table rotate 90 degrees.

Let me show you what I mean by that.

\[ +3 = 65 \text{ min. (Y:45)} \]
For example, consider the control signal RegWrite.

If we treat all the don't cares as zeros, this row here means RegDest has to be equal to one whenever we have a R-type, or an OR immediate, or a load instruction.

Since we can determine whether we have any of these instructions (point to the column headers) by looking at the bits in the “OP” field, we can transform this symbolic equation to this binary logic equation.

For example, the first product term here say we have a R-type instruction whenever all the bits in the “OP” field are zeros.

So each of these big AND gates implements one of the columns (R-type, ori, ...) in our table. Or in more technical terms, each AND gate implements a product term.

In order to finish implementing this logic equation, we have to OR the proper terms together.

In the case of the RegWrite signal, we need to OR the R-type, ORi, and load terms together.

\[ +2 = 67 \text{ min.} \quad (Y:47) \]
Similarly, for ALUSrc, we need to OR the ori, load, and store terms together because we need to assert the ALUSrc signals whenever we have the Ori, load, or store instructions.

The RegDst, MemtoReg, MemWrite, Branch, and Jump signals are very simple. They don’t need to OR any product terms together because each is asserted for only one instruction.

For example, RegDst is asserted ONLY for R-type instruction and MemtoReg is asserted ONLY for load instruction.

ExtOp, on the other hand, needs to be set to 1 for both the load and store instructions so the immediate field is sign extended properly.

Therefore, we need to OR the load and store terms together to form the signal ExtOp.

Finally, we have the ALUop signals.

But clever encoding of the ALUop field, we are able to keep them simple so that no OR gates is needed.

If you don’t already know, this regular structure with an array of AND gates followed by another array of OR gates is called a Programmable Logic Array, or PLA for short.

It is one of the most common ways to implement logic function and there are a lot of CAD tools available to simplify them.

+3 = 70 min. (Y:50)
OK, now that we have the Main Control implemented, we have everything we needed for the single cycle processor and here it is.

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 data path.

Based on the OP field of the instruction, the Main Control of will set the control signals RegDst, ALUSrc, ... etc properly as I showed you earlier using separate slides.

Furthermore, the ALUctr use 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, Or, and so on.

This processor will execute each of the MIPS instruction in the subset in one cycle.

There is, however, a couple of subtle differences between this single-cycle processor and a real MIPS processor in terms of instruction execution.

\[ +2 = 72 \text{ min (Y:52)} \]
First of all, the effect of the load instruction in a real MIPS processor is delayed. That is if you execute a load register R1 here, Register R1 is not updated until the next-next instruction. The very next instruction will still see the old value. This is due to pipelining, which we will cover later. But in our single cycle implementation, all instructions, including the load are completed in one cycle, so the effect of the load is not delayed.

Another effect of pipelining is that the branch instruction in a real MIPS processor is also delayed. That is if we have a "jump 1000" instruction at memory location 0x00, the next instruction we execute is still the instruction at location 0x004. We don’t jump to location 1000 until the next-next instruction. This is called delay branch and we will spend more time talking about it when we talked about pipeline. But for our single cycle implementation, branch is not delayed so if we execute a “jump 1000,” the very next instruction we execute will come from address location 10000.

\[ +2 \text{ = 74 min. (Y:54)} \]
This timing diagram shows the worst case timing of our single cycle datapath which occurs at the load instruction.

Clock to Q time after the clock tick, PC will present its new value to the Instruction memory. After a delay of instruction access time, the instruction bus (Rs, Rt, ...) becomes valid.

Then three things happen in parallel:
(a) First the Control generates the control signals (Delay through Control Logic).
(b) Secondly, the register file access is to put Rs onto busA.
(c) And we have to sign extended the immediate field to get the second operand (busB).

Here I assume register file access takes longer time than doing the sign extension so we have to wait until busA valid before the ALU can start the address calculation (ALU delay).

With the address ready, we access the data memory and after a delay of the Data Memory Access time, busW will be valid.

And by this time, the control unit would have set the RegWr signal to one so at the next clock tick, we will write the new data coming from memory (busW) into the register file.

\[ +3 = 77 \text{ min. (Y:57)} \]
Well, the last slide pretty much illustrates 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 components: Clock to Q time of the PC, 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.

Another problem of this single cycle implementation is that this cycle time, which is long enough for the load instruction, is too long for all other instructions.

We will show you why this is bad and what we can do about it in the next few lectures.

That's all for today.

+2 = 79 min (Y:59)
If you want to find out more information on this topic, you should read Section 5.1 to 5.3 of your text book.

Finally, if you want the official reference on the MIPS architecture, here is the book.

+1 = 80 min. (Z:00)