Computer Science 324
Computer Architecture
Mount Holyoke College
Fall 2007

Topic Notes: Data Paths and Microprogramming

We have spent time looking at the MIPS instruction set architecture and building up components from digital logic primitives. Our next goal is to see how we can use the physical devices we have studied to construct the hardware that can execute MIPS instructions.

A MIPS Subset Implementation

To keep things manageable, we will consider a subset of key MIPS instructions.

1. memory access: `lw`, `sw`
2. arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
3. control flow: `beq`, `j`

The same ideas and techniques that we will use to implement this basic subset can be used to build a machine to implement the entire MIPS ISA, and in fact, most modern ISAs.

Let's think about what needs to be done to implement these instructions.

First, recall the loop that our machine will execute:

1. Fetch the instruction from memory at the location indicated by the program counter
2. Update the program counter to point to the next instruction to be executed
3. Decode the instruction
4. Execute the instruction
5. Go to 1

The first two steps are done the same way, regardless of the instruction we're going to execute. During the decode and execution steps, the implementation becomes instruction-specific.

At that point, the instruction may result in a register value being written into memory, a register value being read from memory, two registers being set as inputs to the ALU and the ALU result written back to another register, or the PC possibly modified by a conditional branch or unconditional jump.

We'll follow the basic implementation in P&H Chapter 5. We start with an abstract view and fill in the details.

The first view is in Figure 5.1.

Let's understand what's in this diagram:

- Functional units:
  - Separate instruction and data memories: this allows the instruction to be fetched and a data value to be read or written from memory in the same instruction cycle
  - Register file: the 32 32-bit registers we saw earlier in the semester
  - Program counter (PC) register
  - Main ALU
  - Two additional adders, one that always adds 4 to the PC (for when we are simply going to advance to the next instruction), and another that computes branch targets
- Data paths:
  - PC gets passed to the instruction memory and to the +4 adder
  - The fetched instruction is decoded and appropriate bits are sent to the input of the second branch target adder (when PC offsets are part of the instruction), to the register file to determine which registers are to be used by the instruction (needed by nearly all instructions) and directly as an ALU input (for immediate mode operands)
  - The result of the PC adders is sent back to update the PC
  - Register file outputs are fed into the ALU and into the data memory
  - Main ALU outputs can determine the address for a main memory access or can be fed back to the register file for storage
  - A value read from the data memory may also be passed back to be stored in the register file

This view is too simplistic for several reasons. In our first refinement of the original abstract diagram, we add some multiplexers and control lines.

This refinement is shown in P&H in Figure 5.2.

What have we added in this refinement?

- Multiplexors replace the "wired or" points in the diagram – those places that two possible inputs come together
  - The MUX at the top selects which value is used to update the PC
  - The MUX whose output goes to the data input of the register file selects between an ALU result and a value read from memory
  - The MUX whose output goes to the main ALU input selects between a second register to the ALU and an immediate value taken from a bit field of the instruction

- Control lines to determine the operation of the individual components
  - The control structure is guided mainly by the instruction, hence the new communication path from the instruction to the `Control` oval
  - That `Control` decodes the instruction and determines which of our functional units are involved in this instruction and what operations they need to perform
  - If the instruction involves storing a value in a register, the `RegWrite` line is set and the value sent to the "Data" input of the register file is stored in the destination register
  - If the instruction is `lw`, the `MemRead` line is set, causing the data memory to retrieve the value at the address computed by the ALU and sends it to the data input of the register file (which also requires that the MUX selects the memory output to be passed to that input)
  - If the instruction is `sw`, the `MemWrite` line is set, causing the value retrieved by the source register to be written to the memory location as determined by the output of the main ALU
  - The main ALU is always computing something, and in those cases that its result is important, a set of control lines tell the ALU which of its functions to compute
  - Finally, if it is a branch instruction, the `Branch` line is set. If the main ALU also produced a zero result (which would cause the ALU to set the `Zero` line), the PC MUX selects the value from the branch target ALU instead of the +4 ALU to be passed to the PC

Building these Components

Our design consists of

1. combinational units – ALUs
2. state elements – memories and registers
3. data signals – information that is stored in memory, registers, or used as inputs and outputs of ALUs
4. control signals – information that controls what the combinational units are to compute, which values should be passed through by the multiplexors, and when state elements should assert their values as data signals (drive the bus) or update their values based on input data signals

All of our components need to be synchronized properly to ensure that inputs and outputs are available at appropriate times.

For example, we have seen flip-flops (and registers built from those flip-flops) that load new values only on the leading edge of the clock. In those cases, we need to make sure that the input is

presented and control lines set appropriately when the clock that controls those flip-flops goes high.

This is sometimes easy – value is ready and available when we need it. Other times, the timing is more subtle. Since combinational units are *always* computing, we need to make sure the input values are presented and the control lines remain correct long enough for the output to be computed and captured.

We consider subsets of the design proposed earlier.

First, the PC and instuction memory.

- Memory is usually thought of as a state element, but the instruction memory is never modified by our simple data path, so it is always producing the instruction value at the location specified on the instruction address (Of course, the program has to get there somehow, so there must be a write capability but we will not consider it for now)
- The PC is just a single register. It can always be writing its value to the instruction address input, and should read a new value at the end of the instruction execute cycle, once we have computed the new PC value
- The adders are combinational units along the lines of those we constructed. One is hardwired to add 4 (and could be replaced with a simpler circuit than a ripple-carry adder if we wanted to save some gates and delay – remember your lab problem).

Next, we consider the implementation of R-format instructions:

```
op $t1, $t2, $t3
```

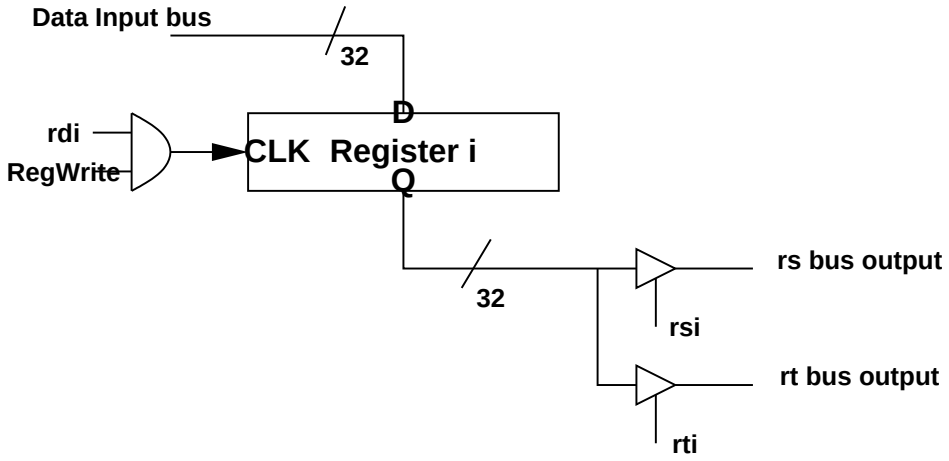
This will write a value to register `$t1` as a result of applying the specified operation on `$t2` and `$t3`.

Thus, we need our register file to be able to produce two output data values and receive one input data value.

We also need to be able to determine which of the 32 registers is to be used to each operand. This information comes directly from the bits of the instruction that specify the two source registers (`rs` and `rt`) and the destination register (`rd`).

To achieve this, we can first decode those 5-bit values using 3 5-to-32 decoders, calling the decoded signals $\texttt{rs}_0$, $\texttt{rs}_1$, ... $\texttt{rs}_{31}$, $\texttt{rt}_0$, $\texttt{rt}_1$, ... $\texttt{rt}_{31}$, $\texttt{rd}_0$, $\texttt{rd}_1$, ... $\texttt{rd}_{31}$.

We can then implement each register i as a 32-bit register:



That takes care of the register file.

What about the main ALU?

Appendix B on the CD with the text describes the ALU construction. We have seen just about everything we need, so we will look at the book's figures to see how an ALU tailored to this MIPS subset can be constructed.

Key ideas:

- construct circuits to compute each needed input
- multiplex the outputs based on an operation selection control line set
- AND and OR are trivial
- we know how to build an adder/subtractor - this ALU works similarly
- Also need `slt` support: set on less than
  - we can tell that $a < b$ if we find that $(a - b) < 0$
  - however, the bit of the ALU that detects whether this value is negative is the high-order (sign) bit, but we want to set the low-order bit in this case
  - Appendix B shows special circuitry needed to accomplish this
  - all bits have a `Less` input, which will be 0 on all but the low-order bit, where it is connected to a copy of the sign bit
- And to support `beq`, we need to have an output that subtracts one of the registers being compared to the other, then checks if the result is zero
- The ALU has 4 control lines but only 6 meaningful combinations, as seen in Figure B.5.13.
  - in this table, the first control line is `Ainvert`, which is only used for the `NOR` functionality

- the second is `Bnegate`, used when we want subtraction (either for the `sub` instruction or because we need the result of a subtraction for `slt`) or `NOR` (where we only care about the "invert" part of the "negate")
- the last two control the multiplexor that selects among the outputs of the AND gate, OR gate, full adder, or `Less`

Implementing Remaining Instructions in our Subset

First, the load and store instructions, which are in the I-format.

```
lw $t0, 1200($t1)
sw $t2, 32($t1)
```

In either case, we need to retrieve the value `$t1` from the register file, and add to that the offset, which is part of the instruction itself. We'll use the main ALU for this. The value computed is the *effective address* for the memory access.

We can't just take the 16 bits from the instruction and add it directly to the contents of the base register. The offset may be negative, so we will need a sign extension unit that will copy the contents of the high order bit of the offset into all higher bits, giving us the 32-bit equivalent.

For a `lw` instruction, we instruct the memory to retrieve the value at the effective address, and it will be stored back in the register file at the destination.

For a `sw` instruction, we need to take the value in the source register from the register file, and present it as input to the memory and tell the memory to write.

This gives the data path shown in Figure 5.10 of P&H (which ignores branch and jump instructions).

For the `beq` instruction, we also need to sign extend the offset value, but then shift it left by 2 before feeding it to the branch target adder. The left shift by 2 accounts for the fact that the branch offset is a number of words, not bytes, that must be added to the `PC+4` value to obtain the branch target location.

Other than that, we need to send the two register values to the ALU to see if they are equal (which we accomplish by testing if the difference produces a 0 result).

The data path for this part is shown in Figure 5.9 of P&H.

If we put together everything we have seen so far, we get the data path of Figure 5.11 of P&H.

This handles all of our instructions except `j`.

Adding Control

Now we want to add the details of the control to the data path.

First, we consider the ALU. We saw that it has 4 control lines. When do we want to set these lines?

This process is a familiar one for us: based on the instruction `opcode` field and (if the opcode indicates an R-format instruction), the `funct` field, we can compute 4 expressions (and hence circuits) that set the ALU control lines appropriately.

Figures 5.12 and 5.13 in P&H show some details of this, but we will not worry about those details at this point.

Next, we consider how the fields of the instruction are used to construct the rest of the needed control signals.

A refinement is shown in Figure 5.15 of P&H.

- Our instruction memory produces the 32-bit instruction word.
- Bits 15-0 (the `address` field for an I-format instruction) are sent to the sign extension unit to be used as potential input values by the ALU.
- Bits 5-0 (the `funct` field for an R-format instruction) are sent to the ALU control to compute the appropriate ALU control lines.
- Bits 25-21 (the source register `rs`) are sent to the register file to select read register 1.
- Bits 20-16 (the source register `rt`) are sent to the register file to select read register 2.
- The write register is more complex. For `lw`, we use the `rt` field in bits 20-16. For R-format instructions, we use the `rd` field in bits 15-11. An additional multiplexor and a control line `RegDst` control which field is passed to the write register selection input.
- The other control lines are computed from the `opcode` in bits 31-26, as shown in Figure 5.17 of P&H. The details of the conversion of the `opcode` are just combinational logic, which again, we can figure out (or look up in the text).

P&H has a series of figures (5.19, 5.20, and 5.21) that show how each type of instruction uses this data path.

Adding Jump

The final refinement is to add the data path and control to implement the `j` instruction, as seen in Figure 5.24.

Using Multiple Cycles to Implement Instructions

The design we've been studying is a "single-cycle" implementation – meaning that one clock cycle results in one instruction being executed.

This is not used in real life, mainly because of the inefficiency:

- Every instruction takes the same amount of time – we don't make the common case fast

- We have redundant elements: the two memory systems, multiple ALU/adder units

If we break our instructions down to operate over a series of (shorter) clock cycles, we can use only the number of cycles we need and potentially reuse some components.

We will develop a machine that has a single memory for both data and instructions (which means we will need to access it more than once per instruction sometimes), and a single ALU (which will need to compute more than one thing during an instruction sometimes).

We will go through a similar refinement process for a multicycle machine that we did for the single cycle machine.

The basic data path is shown in Figure 5.26 of P&H.

- Here we see that there is just the one memory and one ALU
- Several registers (that are not part of the ISA and hence not visible to a programmer) have been added that will store values that will no longer be readily available when we need them
  - An instruction register (IR) that gets loaded from memory with the currently-executing instruction
  - A memory data register that holds non-instruction values that have been read from memory
  - Registers A and B that hold values read from the register file that are to be used as ALU inputs
  - A register to hold the result of the ALU
- The ALU now needs to be able to add values from a register or the PC to a value from a register or the constant 4 (when computing PC+4), or the `address` field from the instruction (when computing an effective address for `lw` or `sw`) or the shifted value of the `address` field (for a branch)

In Figure 5.27, the needed control inputs are added.

- Some control lines remain from our previous design: `MemRead`, `MemWrite`, `MemtoReg`, `ALUOp`, `RegWrite`, `RegDst`
- Some new ones:
  - `IorD`: are we loading an instruction (using the PC as an address) or data (using the ALU output as an address)
  - `IRWrite`: when to copy in the memory data output to the IR
  - `ALUSrcA`: is the first ALU input coming from a register via the A register or the PC

  - `ALUSrcB`: is the second ALU input coming from a register via the B register, the constant, 4, the sign-extended raw `address` field from the instruction, or that value shifted left by 2 (2 bits needed for this control)

Figure 5.28 shows the needed control lines (including jumps).

- The main control at the top takes the `opcode` part of the instruction as input
- Note that the `j` instruction is supported here, by the extra Mux that selects the next PC value among the current ALU result, an ALU result stored in ALUOut, or the jump address taken from a `j` instruction

---

Control for Multicycle Execution

Unlike the single cycle implementation, we need to turn on and off control lines in sequence during the execution of an instruction.

Idea: break down the instruction into substeps, each of which can be done in a clock cycle.

- Substeps should be as close as possible to each other in terms of time required – the longest will determine how fast the clock can run
- a substep is restricted to a single memory access, register file access, or ALU operation (zero or one of each) – to keep things fast
- there is no harm in using more than one of these components in a subset – they can operate in parallel!
- edge-triggered methodology – we capture results of ALU operations, register file access, and memory accesses into internal registers to be used in subsequent phases

We break down the execution into these subphases. Each instruction uses 3 to 5 of these subphases.

1. **Instruction fetch**:

   ```
   IR <= Memory[PC];
   PC <= PC + 4;
   ```

   Read the `<=` as "gets" – the `IR` gets the result of accessing memory at the location specified by the `PC`.

   So in this subphase, we need to

   - set `MemRead` to 1 to read memory

- set `IRWrite` to 1 to capture this into the `IR`
- set `IorD` to 0 to take the memory address from the `PC`
- set `ALUSrcA` to 0 to pass the `PC` as an ALU input
- set `ALUSrcB` to 01 to pass the constant 4 as an ALU input
- set `ALUop` to 00 to make the ALU perform addition
- set `PCSource` to 00 to pass the `PC+4` value back to the `PC`
- set `PCWrite` to 1 to copy this value back into the `PC`

That's a lot of work, but no component is being used more than once.

One concern might be that the `PC` gets rewritten too soon – we need to make sure the `IR` has been loaded, but that will be the case since the same edge trigger will copy the value from memory into the `IR` and the updated `PC` into the `PC`.

Another concern might be that the `PC` gets loaded with `PC+4` before we have a chance to determine that the instruction might be `beq` or `j`. In those cases, we will subsequently overwrite the `PC` before this subphase comes around again and the `PC` is used to load the `IR` with the next instruction.

2. **Instruction decode and register fetch**:

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend (IR[15:0]) << 2);
```

Here, we continue operations that either need to or can happen (without ill effect), regardless of the instruction.

Many instructions need to use subfields `rs` and `rt` to select values from the registers, so we retrieve those into our internal registers `A` and `B`.

The final part is the computation of a potential branch target address, which we may or may not need. However, the ALU isn't doing anything else yet in this subphase, so we will use it to compute this "just in case".

Here, our control needs to:

- set `ALUSrcA` to 0, sending `PC` as the first ALU input
- set `ALUSrcB` to 11, sending the sign-extended and shifted `offset` field to the second ALU input
- set `ALUop` to 00 to request addition

Everything else happens anyway!

- `A` and `B` are unconditionally loaded from the register file on every subcycle
- `ALUOut` is loaded with the result of the ALU on every subcycle

3. **Execute ALU op/compute effective address/complete branch**

   Here, we finally do things that depend on the instruction being executed.

   There are four possibilities:

   Memory In this case, we simply want to compute the effective address:

   ```
   ALUOut <= A + sign-extend (IR[15:0]);
   ```

   A already contains the value from the register specified by `rs`. So we add that to the sign-extended offset from the `IR`.

   The control specification:

   - set `ALUSrcA` to 1, sending `A` as the first input
   - set `ALUSrcB` to 10, sending the sign-extended `offset` bits as the second ALU input
   - set `ALUOp` to 00 to request addition

   Our next step here will be to access memory, but we can't do that until the next cycle since we will not have the effective address in `ALUOut` until the end of this cycle.

   R-type For R-format instructions, we are set to perform the requested ALU operation:

   ```
   ALUOut <= A op B;
   ```

   `A` and `B` have the values from the appropriate registers after cycle 2. We now combine them according to the operation specified by the `funct` field of the instruction.

   We can imagine taking this result and writing it directly back to the register file in this cycle, but that would require too much time and would slow down other operations. So we just capture the ALU result in `ALUOut` to be stored in our next cycle.

   The control specification:

   - set `ALUSrcA` to 1, sending `A` as the first input
   - set `ALUSrcB` to 00, sending `B` as the second ALU input
   - set `ALUOp` to 10 to use the `funct` field to select the correct ALU operation

   Branch For a branch instruction, we just computed a potential branch target and stored it in `ALUOut`. Now we need to do the comparison to decide if this should be stored in the `PC`.

   ```
   if (A == B) PC <= ALUOut;
   ```

   To accomplish this, we subtract to set the `Zero` flag of the ALU. If `Zero` is set, we want to copy `ALUOut` into the `PC`.

   The control specification:

   - set `ALUSrcA` to 1, sending `A` as the first input
   - set `ALUSrcB` to 00, sending `B` as the second ALU input
   - set `ALUOp` to 01 to request subtraction

- set `PCSource` to 01 to send the contents of `ALUOut` as an input value to the `PC` register
- set `PCCondWrite` to 1 which will cause the `PC` to be updated with the value from its input only if `Zero` is set by the ALU

Note that `ALUOut` will be updated to contain the new ALU result, overwriting our branch target. This is fine, though, since that value will be copied out into the `PC` "just in time" before it gets overwritten. The timing is crucial here.

This completes the branch instruction (in 3 cycles). We can go back to step 1 and fetch the next instruction.

Jump If the instruction is a branch, we can also wrap it up on this cycle.

```
PC <= { PC[31:28], IR[25:0], 00) };
```

This is just a way to say that the new value of the `PC` is constructed using the top 4 bits of the old `PC`, followed by the 26 bits of the jump instruction that form the target, followed by 2 0 bits.

The control specification here is very simple:

- set `PCSource` to 10 to select the 32-bit value constructed as above (by wired connections)
- set `PCWrite` to 1 to copy in this new value to the `PC`

The jump is done, and we can go back to step 1 on the next cycle.

4. **Access memory/write back ALU result**

Both memory access and R-format instructions will require this step.

R-type For R-format instructions, the previous phase computed the result and stored it in `ALUOut`. We now need only copy it back into the appropriate destination register.

```
Reg[IR[15:11]] <= ALUOut;
```

The control to accomplish this:

- set `RegDst` to 1 to use the `rd` field to sety the write register number
- set `MemToReg` to 0 to place the value of `ALUOut` onto the data input of the register file
- set `RegWrite` to copy the value from the data input into the selected write register

Our R-type instruction is complete, and we can continue with the instruction fetch on the next cycle.

Memory Read For memory access instructions, we have computed an effective address and stored it in `ALUOut`, and we use this as the address input of our memory unit.

For reading, we need to capture the value from memory to be written to the register file.

```
MDR <= Memory[ALUOut];
```

The control:

- set `IorD` to 1 to pass the value in `ALUOut` to the memory address inputs
- set `MemRead` to read from memory

The `MDR` always captures the value from the MemData output, so we don't need to do anything specific there.

One note here: we might be tempted to wire the MemData output directly to the Mux that selects the input to the Write Data input on the register file. We choose not to do so because the register write operation would need to be delayed long enough for the memory access to complete. Memory access is already the slowest individual operation in our system, so we will avoid anything else in the same cycle that cannot be done in parallel with the memory access.

Memory Write As with the read operation, the `ALUOut` register contains the effective address.

Meanwhile, the `B` register contains the value of the source register from the register file that we wish to store in memory. We didn't specifically worry about this up to this point, but that value is in fact copied from the register specified by bits 20:16 of the `IR` during every cycle. So our operation is simply:

```
Memory[ALUOut] <= B;
```

The control to achieve this:

- set `IorD` to 1 to pass the value in `ALUOut` to the memory address inputs
- set `MemWrite` to 1 to read the value from `B` into memory

That's it for memory write, and we can continue with the fetch of the next instruction.

5. **Complete memory read**

The only instruction that can get to this cycle is a memory read. In this case, the only work that remains is to copy the value in the `MDR` into the appropriate register in the register file.

```
Reg[IR[20:16]] <= MDR;
```

To accomplish this:

- set `MemtoReg` to 1 to place the `MDR` value on the data input of the register file
- set `RegDst` to 0 to select the `rd` field of this I-format instruction as the register number to be written
- set `RegWrite` to 1 to copy the value into the selected register

Finally, our memory read is complete, after 5 cycles.

Multicycle Control Strategy

P&H describes a methodology for controlling this kind of system using a *finite state machine*.

The idea is that we have a series of *states* that determine the status of the control lines at any given time. When progressing from one cycle to the next, we follow a *transition* to a new state. The new state depends on the current state and the instruction being executed.

The finite state machine we just developed for the multi-cycle architecture is in P&H Figure 5.37.

The implementation of such a system is another example of combinational logic that we can quite easily understand.

- the datapath control outputs are a function of the current state and the instruction opcode
- the next state is also a function of the same input values
- when moving to the next clock cycle to continue execution, the next state becomes the current state

Microprogramming

Another approach, more commonly used for more complex ISAs than MIPS, is a *microprogrammed* design.

While a finite state control might be reasonable for an ISA with a limited instruction set, such a design might result in thousands of states and transitions.

Another way to think about this is to have a program that executes instructions that determine which control lines are activated. In this case, we would have:

- A *microarchitecture*, that includes the data path and control like the multi-cycle MIPS implementation we've been looking at
- A *microsequencer* that runs a *microprogram* made up of *microinstructions*
- Each microinstruction, analogous to a regular machine instruction, would do something (in this case, provide the appropriate control signals to the microarchitecture) then move on to the next instruction, which may be the next in sequence, or may be at the target of a microprogram branch
- The microprogram interprets the machine instructions to be executed, and executes them, step-by-step, by asserting the correct control signals in the correct sequence
- A microprogram can be (and often is) used to implement a more complex ISA on top of a relatively simple microarchitecture

- There may be many ways to implement a complex ISA on a given microarchitecture, and the overriding goal is usually **speed**
- A well-designed microprogram will implement the ISA instructions in as few microinstructions as possible – try to exploit parallelism
- A microprogram can also be used to implement a variety of ISAs on the same microarchitecture
- A microprogramming environment, including a *microassembler* would be used to generate microinstructions from a *microassembly language*

So how to we do all of this?

- Define a microinstruction format – given a microinstruction, what does it mean in terms of the control signals
- Develop a microprogram that, when executed, implements the desired instruction set architecture on the given microarchitecture.

One possible microcode approach is described in P&H Section 5.7 on the CD.

My implementation of a microcode simulator that approximates the book's approach is in the Lab 6 starter code.

- My microinstructions consist of several fields, as described in the file `ucode.format` in the lab 6 starter
  - while the microprogram for our small subset of MIPS fits in only 10 microinstructions, I designed the microinstruction format to allow a microcode store of 256 microinstructions
  - given all of the control lines we need to manage, a 2-bit sequencing indicator, and the 8 bits for a microinstruction address, we need a total of 26 bits for a microinstruction
  - the simulator uses a 32-bit value for this, but in a real machine, we'd just build a ROM with the correct number of bits to hold microinstructions – no need to make it a power of 2
- All control line values can be obtained directly from the appropriate bits in the current microinstruction
- The sequencing control in this case is where this approach gains its power – possibilities are:
  - continue to the next microinstruction in sequence
  - branch unconditionally to a new microinstruction (note: the first can be a special case of this)
  - branch conditionally based on opcode using a dispatch table