SIENA*college*
Computer Science

Computer Science 220
Assembly Language & Comp. Architecture
Siena College
Fall 2011

Topic Notes: Data Paths and Control

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 4. We start with an abstract view and fill in the details.

The first view is in Figure 4.1.

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 4.2.

What have we added in this refinement?

- Multiplexers 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 multiplexers, 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 binary incrementer problem).

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

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

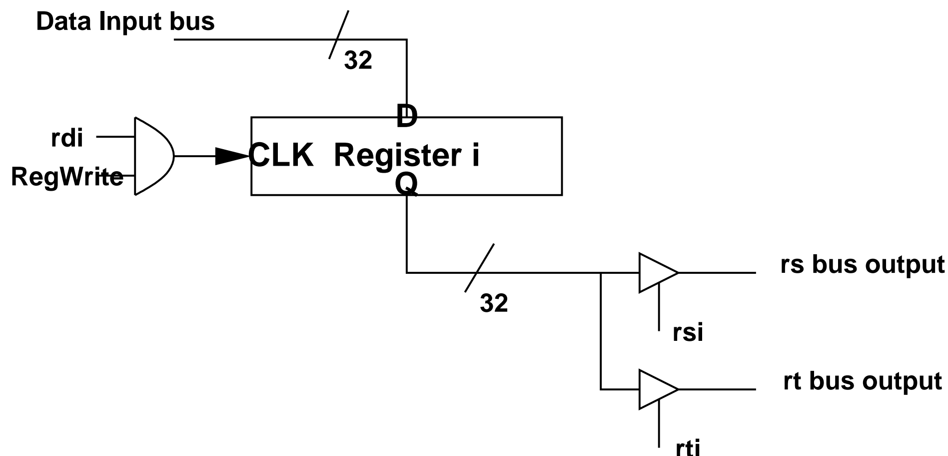
This will write a value to register `$t1` as a result of applying the specified operation on `$t2` and `$t3`.

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 consisting of 32 D-type flip-flops (for example):



That takes care of the register file.

What about the main ALU?

Appendix C 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 C 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 C.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 multiplexer 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 4.10 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 4.9 of P&H.

If we put together everything we have seen so far, we get the data path of Figure 4.11 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 4.12 and 4.13 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 4.15 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 multiplexer 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 4.17 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 (4.19, 4.20, and 4.21) 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 4.24.

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.

There are a number of ways to do this – we will consider a common approach called pipelining.