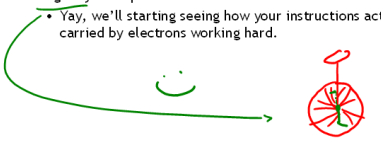
Lecture 8

- Announcement:
  - HW1 due today. Hope you had fun!
    - Remember, 2 late days only so we can post solution and talk about it
  - HW2 out. Functions, stack smashing ☺.
- Today:
  - Assembly and machine language wrap-up
  - Single-cycle implementation
    - Yay, we'll starting seeing how your instructions actually gets carried by electrons working hard.



1

Assembly Language Wrap-Up

We've introduced MIPS assembly language

Remember these ten facts about it

1. MIPS is representative of all assembly languages - you should be able to learn any other easily
2. Assembly language is machine language expressed in symbolic form, using decimal and naming
3. R-type instruction op $r1,$r2,$r3 is $r1 = $r2 op $r3
4. I-type instruction op $r1,$r2,imm is $r1 = $r2 op imm
5. I-type is used for arithmetic, branches, load & store, so the roles of the fields change
6. Moving data to/from memory uses imm($rs) for the *effective address*, ea = imm + $rs, to reference M[ea]

2

Ten Facts Continued

7. Branch and Jump destinations *in instructions* refer to words (instructions) not bytes
8. Branch offsets are relative to PC+4
9. By convention registers are used in a disciplined way; following it is wise!
10. "Short form" explanation is on the green card, "Long form" is in appendix B

3

Instruction Format Review

Register-to-register arithmetic instructions are R-type

op	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Load, store, branch, & immediate instructions are I-type

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

The jump instruction uses the J-type instruction format

op	address
6 bits	26 bits

4

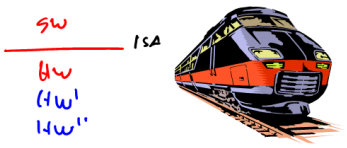
Review

- If three instructions have opcodes 1, 7 and 15 are they all of the same type?
- If we were to add an instruction to MIPS of the form MOD $t1, $t2, $t3, which performs $t1 = $t2 MOD $t3, what would be its opcode?
- How can you tell if the immediate field is positive or negative?
- Could the distance J jumps be increased by using an opcode of fewer bits?

5

A single-cycle MIPS processor

- An instruction set architecture is an *interface* that defines the hardware operations which are available to software.
- Any instruction set can be implemented in many different ways. Over the next few weeks we'll see several possibilities.
  - In a basic single-cycle implementation all operations take the same amount of time—a single cycle.
  - A multicycle implementation allows faster operations to take less time than slower ones, so overall performance can be increased.
  - Finally, pipelining lets a processor overlap the execution of several instructions, potentially leading to big performance gains.



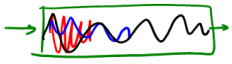
6

Single-cycle implementation

- We will describe the implementation a simple MIPS-based instruction set supporting just the following operations.

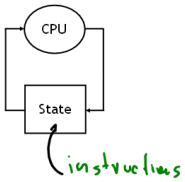
Arithmetic:	add	sub	and	or	slt
Data Transfer:	lw	sw			
Control:	beq				

- Today we'll start building a single-cycle implementation of this instruction set.
  - All instructions will execute in the same amount of time; this will determine the clock cycle time for our performance equations.
  - We'll explain the datapath first, and then make the control unit.



Computers are state machines

- A computer is just a big fancy state machine.
  - Registers, memory, hard disks and other storage form the state.
  - The processor keeps reading and updating the state, according to the instructions in some program.

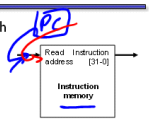


John von Neumann

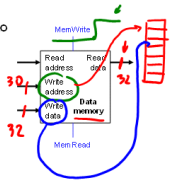
- In the old days, "programming" involved actually changing a machine's physical configuration by flipping switches or connecting wires.
  - A computer could run just one program at a time.
  - Memory only stored data that was being operated on.
- Then around 1944, John von Neumann and others got the idea to encode instructions in a format that could be stored in memory just like data.
  - The processor interprets and executes instructions from memory.
  - One machine could perform many different tasks, just by loading different programs into memory.
  - The "stored program" design is often called a Von Neumann machine.

Memories

- It's easier to use a Harvard architecture at first, with programs and data stored in *separate* memories.
- To fetch instructions and read & write words, we need these memories to be 32-bits wide (buses are represented by dark lines here). We still want byte addressability, so these are 2^{30} x 32 memories.

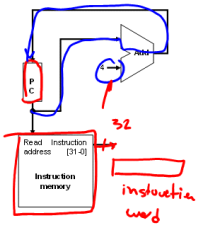


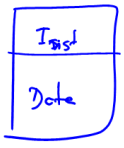
- Blue lines represent control signals. MemRead and MemWrite should be set to 1 if the data memory is to be read or written respectively, and 0 otherwise.
  - When a control signal does something when it is set to 1, we call it active high (vs. active low) because 1 is usually a higher voltage than 0.
- For now, we will assume you cannot write to the instruction memory.
  - Pretend it's already loaded with a program, which doesn't change while it's running.



Instruction fetching

- The CPU is always in an infinite loop, fetching instructions from memory and executing them.
- The program counter or PC register holds the address of the current instruction.
- MIPS instructions are each four bytes long, so the PC should be incremented by four to read the next instruction in sequence.



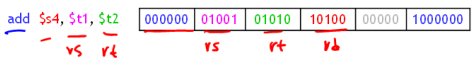


Encoding R-type instructions

- Last lecture, we saw encodings of MIPS instructions as 32-bit values.
- Register-to-register arithmetic instructions use the R-type format.
  - op is the instruction opcode, and func specifies a particular arithmetic operation (see textbook).
  - rs, rt and rd are source and destination registers.

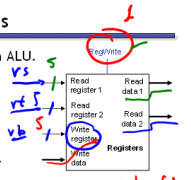
op	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- An example instruction and its encoding:



Registers and ALUs

- R-type instructions must access registers and an ALU.
- Our register file stores thirty-two 32-bit values.
  - Each register specifier is 5 bits long.
  - You can read from two registers at a time.
  - RegWrite is 1 if a register should be written.

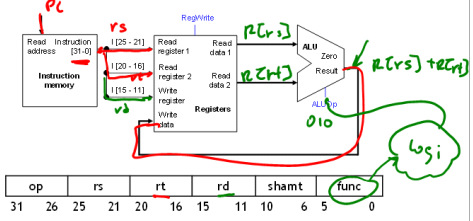


- Here's a simple ALU with five operations, selected by a 3-bit control signal ALUOp.

ALUOp	Function
000	and
001	or
010	add
110	subtract
111	slt

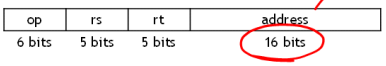
Executing an R-type instruction

1. Read an instruction from the instruction memory.
2. The source registers, specified by instruction fields rs and rt, should be read from the register file.
3. The ALU performs the desired operation.
4. Its result is stored in the destination register, which is specified by field rd of the instruction word.



Encoding I-type instructions

- The lw, sw and beq instructions all use the I-type encoding.
  - rt is the *destination* for lw, but a *source* for beq and sw.
  - address is a 16-bit signed constant.

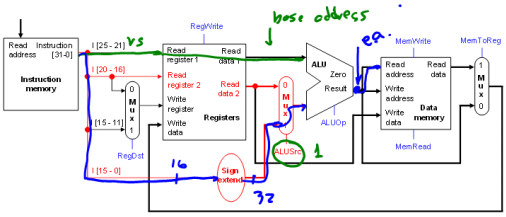


- Two example instructions:

lw	$t0, -4($sp)	100011	11101	01000	1111 1111 1111 1100
sw	$a0, 16($sp)	101011	11101	00100	0000 0000 0001 0000

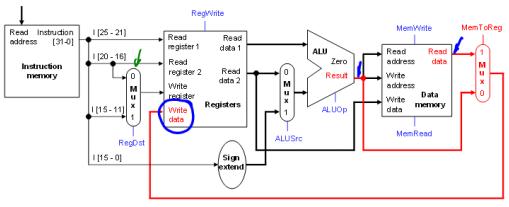
Accessing data memory

- For an instruction like lw $t0, -4($sp), the base register $sp is added to the *sign-extended* constant to get a data memory address.
- This means the ALU must accept *either* a register operand for arithmetic instructions, *or* a sign-extended immediate operand for lw and sw.
- We'll add a multiplexer, controlled by ALUSrc, to select either a register operand (0) or a constant operand (1).



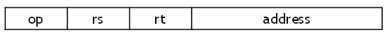
MemToReg

- The register file's "Write data" input has a similar problem. It must be able to store *either* the ALU output of R-type instructions, *or* the data memory output for lw.
- We add a mux, controlled by MemToReg, to select between saving the ALU result (0) or the data memory output (1) to the registers.



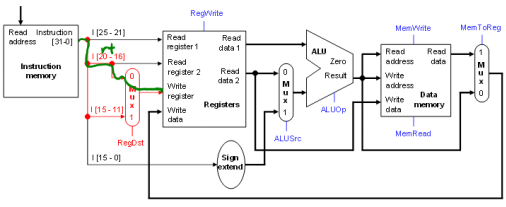
RegDst

- A final annoyance is the destination register of lw is in *rt* instead of rd.



lw $rt, address($rs)

- We'll add one more mux, controlled by RegDst, to select the destination register from either instruction field rt (0) or field rd (1).

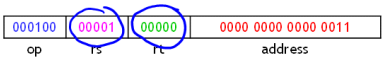


Branches

- For branch instructions, the constant is not an address but an *instruction offset* from the current program counter to the desired address.

```
        beq  $at, $0, L
        add  $v1, $v0, $0
        add  $v1, $v1, $v1
        j    Somewhere
L:      add  $v1, $v0, $v0
```

- The target address L is three *instructions* past the beq, so the encoding of the branch instruction has 0000 0000 0000 0011 for the address field.

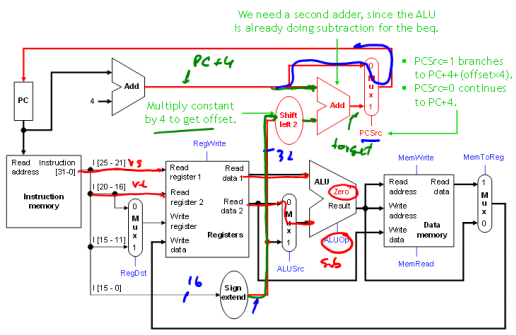


- Instructions are four bytes long, so the actual memory offset is 12 bytes.

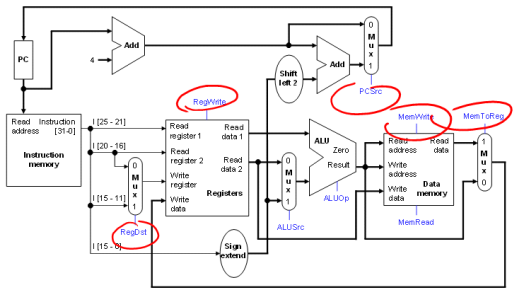
The steps in executing a beq

1. Fetch the instruction, like beq $at, $0, offset, from memory.
2. Read the source registers, $at and $0, from the register file.
3. Compare the values by subtracting them in the ALU.
4. If the subtraction result is 0, the source operands were equal and the PC should be loaded with the target address, PC + 4 + (offset x 4).
5. Otherwise the branch should not be taken, and the PC should just be incremented to PC + 4 to fetch the next instruction sequentially.

Branching hardware



The final datapath

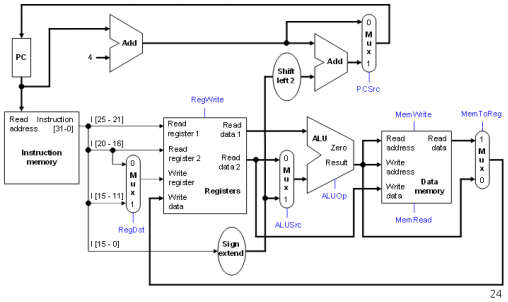


Control

- The control unit is responsible for setting all the control signals so that each instruction is executed properly.
  - The control unit's input is the 32-bit instruction word.
  - The outputs are values for the blue control signals in the datapath.
- Most of the signals can be generated from the instruction opcode alone, and not the entire 32-bit word.
- To illustrate the relevant control signals, we will show the route that is taken through the datapath by R-type, lw, sw and beq instructions.

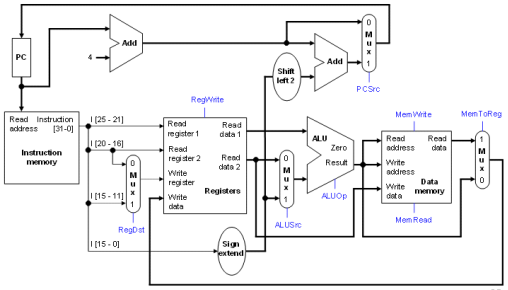
R-type instruction path

- The R-type instructions include add, sub, and, or, and slt.
- The ALUOp is determined by the instruction's "func" field.



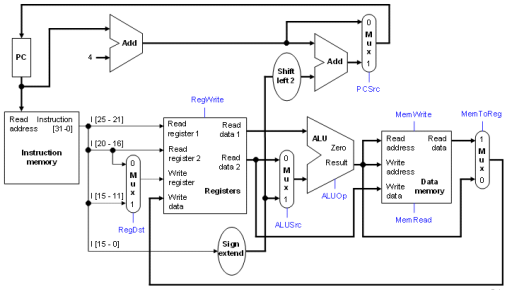
lw instruction path

- An example load instruction is lw $t0, -4($sp).
- The ALUOp must be 010 (add), to compute the effective address.



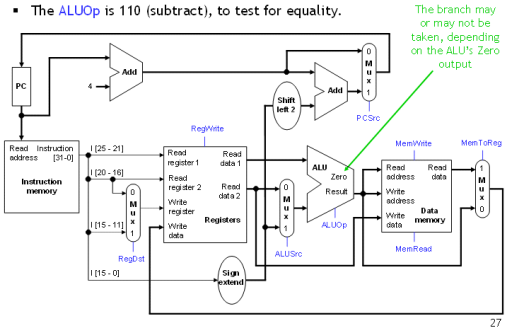
sw instruction path

- An example store instruction is sw $a0, 16($sp).
- The ALUOp must be 010 (add), again to compute the effective address.



beq instruction path

- One sample branch instruction is beq $at, $0, offset.
- The ALUOp is 110 (subtract), to test for equality.



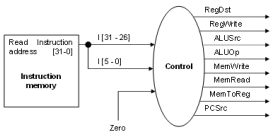
Control signal table

Operation	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemRead	MemToReg
add	1	1	0	010	0	0	0
sub	1	1	0	110	0	0	0
and	1	1	0	000	0	0	0
or	1	1	0	001	0	0	0
slt	1	1	0	111	0	0	0
lw	0	1	1	010	0	1	1
sw	X	0	1	010	1	0	X
beq	X	0	0	110	0	0	X

- sw and beq are the only instructions that do not write any registers.
- lw and sw are the only instructions that use the constant field. They also depend on the ALU to compute the effective memory address.
- ALUOp for R-type instructions depends on the instructions' func field.
- The PCSrc control signal (not listed) should be set if the instruction is beq *and* the ALU's Zero output is true.

Generating control signals

- The control unit needs 13 bits of inputs.
  - Six bits make up the instruction's opcode.
  - Six bits come from the instruction's func field.
  - It also needs the Zero output of the ALU.
- The control unit generates 10 bits of output, corresponding to the signals mentioned on the previous page.
- You can build the actual circuit by using big K-maps, big Boolean algebra, or big circuit design programs.
- The textbook presents a slightly different control unit.



Summary

- A datapath contains all the functional units and connections necessary to implement an instruction set architecture.
  - For our single-cycle implementation, we use two separate memories, an ALU, some extra adders, and lots of multiplexers.
  - MIPS is a 32-bit machine, so most of the buses are 32-bits wide.
- The control unit tells the datapath what to do, based on the instruction that's currently being executed.
  - Our processor has ten control signals that regulate the datapath.
  - The control signals can be generated by a combinational circuit with the instruction's 32-bit binary encoding as input.
- Next, we'll see the performance limitations of this single-cycle machine and try to improve upon it.