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?

1

A single-cycle MIPS processor

An instruction set architecture is an *interface* that defines the hardware operations 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

2

Single-cycle implementation

We describe the implementation of 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 build a single-cycle implementation of this instruction set

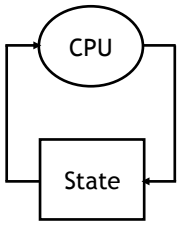
- All instructions will execute in the same amount of time; this will determine the clock cycle time for our performance equations
- We'll explain the datapath first, and then make the control unit

3

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



4

John von Neumann

In ancient times, “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 Atanasoff, Eckert and Machley (and others) got the idea to encode instructions in a format that could be stored in memory just like data

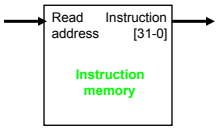
- The processor interprets and executes instructions from memory
- One machine could perform many different tasks, just by loading different programs into memory
- John von Neumann wrote the first explanation of their idea, so the “stored program” design is often called a Von Neumann machine

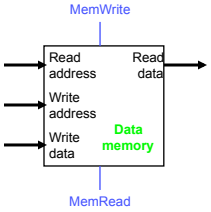
5

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



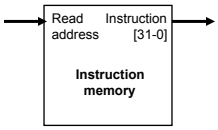


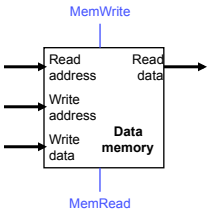
6

More 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





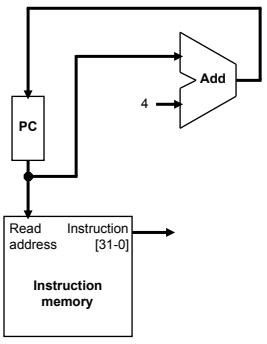
7

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



8

Start With 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
- 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:

add $s4, $t1, $t2

000000	01001	01010	10100	00000	1000000

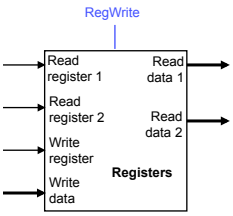
9

Registers and ALUs

R-type instructions must access registers and an ALU

The 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



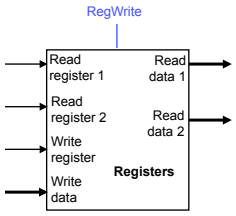
10

Registers and ALUs

R-type instructions must access registers and an ALU

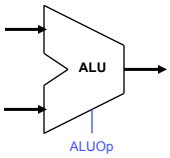
The register file stores thirty-two 32-bit values

- ❖ 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

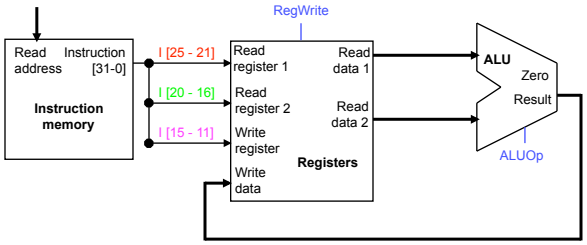
ALUOp	Function
000	and
001	or
010	add
110	subtract
111	slt

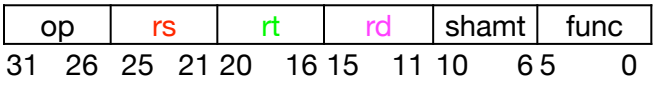


11

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





12

Encoding I-type instructions

The lw, sw and beq instructions are all I-type encoding

- rt is the *destination* for lw, but a *source* for beq and sw
- address is a 16-bit signed constant

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

Two example instructions:

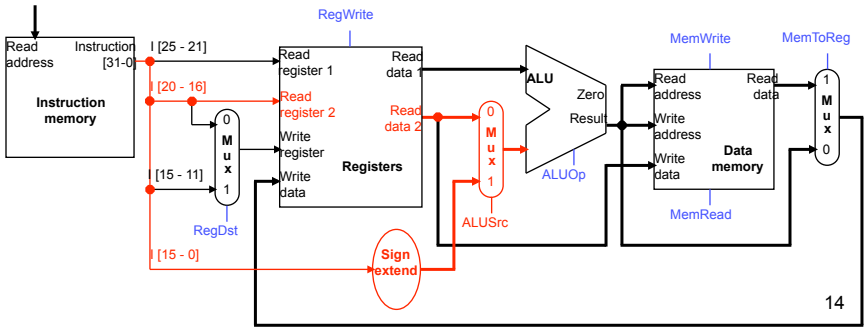
lw $t0, –4($sp)	100011	11101	01000	1111 1111 1111 1100
sw $a0, 16($sp)	101011	11101	00100	0000 0000 0001 0000

13

Accessing data memory

For lw $t0, –4($sp), the base register $sp is added to a *sign-extended* constant to get a data memory address

- So 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)

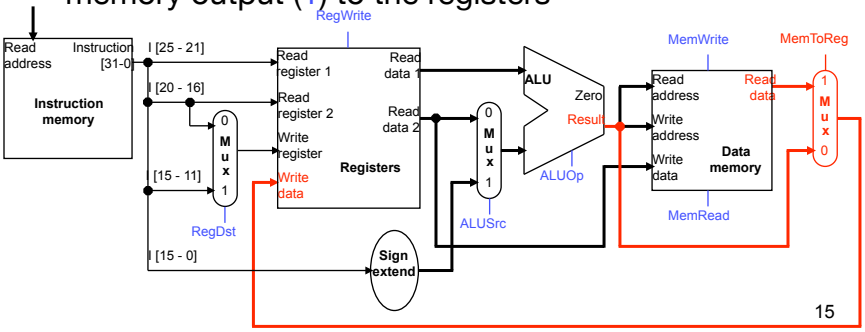


14

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



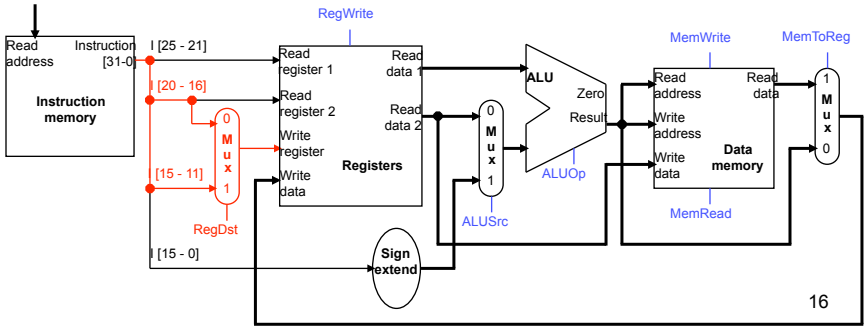
RegDst

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

op	rs	rt	address

lw $rt, address($rs)

We add one more mux, controlled by RegDst, to select the destination register from either the rt (0) or rd (1)

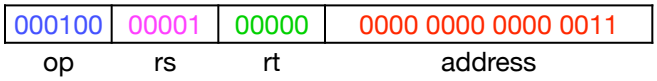


Branches

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.

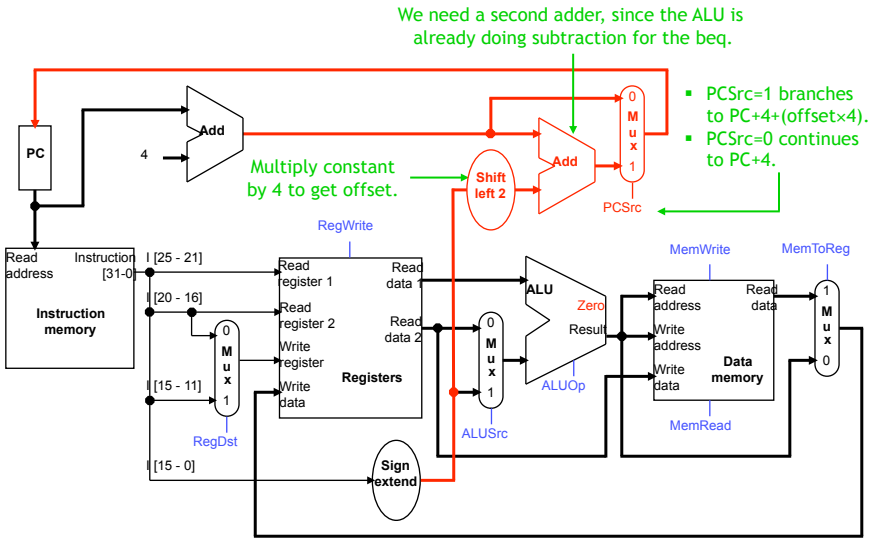
17

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

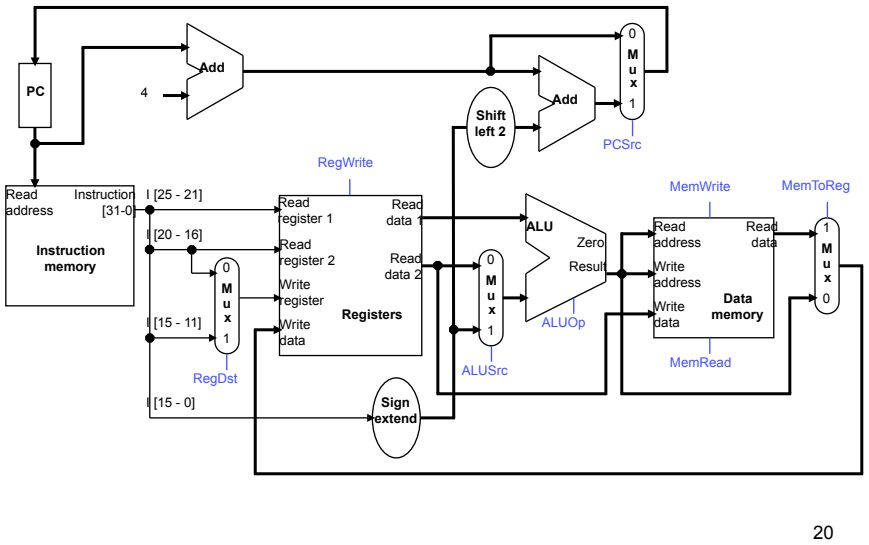
18

Branching hardware



19

The final datapath



20