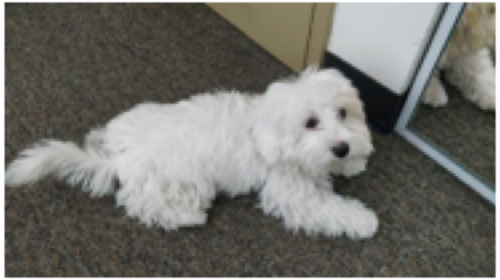
Welcome minute : Shanti!

The words means "Peace"

I love Playing Badminton and Table Tennis

Hate Animal cruelty. I eat lots of veggies, yay!



I miss my dog back home, so I babysit this cutie here

I love dancing and trekking along "short" forest trails :P

🎶 Yellow Submarine, The Beatles 🎶

Announcements

- Send your welcome slide!
- Homework 1 is posted
  - Logistical updates & questions
    - Posted on Gradescope – is it helpful to create an "assignment" on Canvas as well?
    - "Call for Consistency": Homework every week, assigned Thursday, due Thursday
- Reminder: First participation quiz will go live today
  - Due: Tuesday (midnight)
  - You have "something 141" every day (M/W/F lecture; Tu Mini-quiz; Th HW)

Canvas Gradescope

"Control Flow" describes how programs execute

- Jumps
- Procedure call (jump subroutine)
- Conditional Branch
  - Used to implement, for example, if-then-else logic, loops, etc.

- Control flow must specify two things
  - Condition under which the jump or branch is taken
  - If take, the location to read the next instruction from ("target")

Jumps are unconditional control flow.
What do they look like in MIPS?

- need to be able to jump to an absolute address sometimes
- need to be able to do procedure calls and returns

function

	6	26
Jump J-type	**opcode**	**target**

- Jump `j 10000 => PC = 10000`

2^{26}

- Jump and Link `jal 20000 => $31 = PC + 4, and PC = 20000`
  - used for procedure calls

$t1, $t2

add $31

- Jump register `jr $31 => PC = $31`
  - used for returns, but can be useful for lots of other things
  - Q: how to encode `jr` instruction?

R-type add r1, r2, r3

Warning: Some ISAs call jumps "unconditional branches" – useful not to for MIPS

What if we want to condition the control flow? Branches.

```
do { … ; a++; } while (a < 100);
```

#1
→ addi t27, $zero, #1
bne $t, t27, else

- `beq` and `bne` are the only branches you need
  - `beq r1, r2, addr   =>   if (r1 == r2):   goto addr`
- But other operations can be combined…
  - `slt $1, $2, $3     =>   if ($2 < $3) $1 = 1; else $1 = 0`
- `beq`, `bne`, `slt`, `and` `$zero`, can implement all fundamental conditions
  - Always, never, !=, = =, >, <=, >=, <, >(unsigned), <= (unsigned), …

i:j
t=1

regs
$i $j
$w

```
if (i<j)
    w = w+1;
else
    w = 5;
after...
```

⇒

```
          → slt $t $i $j
R-type    → beq $t $0, else      i:j
I-type    → addi $w, $w, 1
          → j after
else:       addi $w, $zero, 5
after:
```

Re-working this example

```
if (i<j)
if_body:
      w = w+1;
else
else_body:
      w = 5;
after_else:
```

```
slt $temp, $i, $j
beq $temp, $zero, else_body
if_body:
addi $w, $w, 1
j   after_else
else_body:
addi $w, $zero, 5
after_else:
```

1. Need to do the comparison
   - Use "store less than", `slt $temp, $i, $j`
     - This writes 1 in $temp when the condition is true
2. Need to decide whether to branch, using only registers
   - Only have `$zero` available to compare with
   - The question is "should we jump over the if body"
   - Want to jump to `else_body` when `$temp` is `0`
   - So we conceptually we are asking `if !(i<j)` [confusing!]
   - `beq $temp, $zero, else_body`
   - This says goto the else body when the `slt` was not true
3. Need to jump over the else body
   - Don't do both the *if* and the *else* on accident!
   - Use "unconditional jump"
   - `j   after_else`
4. Finally, fill in the bodies

FAQs / Extras

```
if (i<j)
if_body:
      w = w+1;
else
else_body:
      w = 5;
after_else:
```

1. Could we have used a `bne` instead?
   - Yes, if you get the value 1 into a register

```
slt  $temp, $i, $j
addi $scratch, $zero, 1
bne  $temp, $scratch, else_body
if_body:
addi $w, $w, 1
j    after_else
else_body:
addi $w, $zero, 5
after_else:
```

   - But this is inefficient
     - Extra instruction
     - *Register pressure*

FAQs / Extras

```
if (i<j)
if_body:
     w = w+1;
else
else_body:
     w = 5;
after_else:
```

1. Could we have used a `bne` with no more instructions?
   - Yes… if you flip the body and "put the else first"

```
slt $temp, $i, $j
bne $temp, $zero, if_body
else_body:
addi $w, $zero, 5
j    after
if_body:
addi $w, $w, 1
after:
```

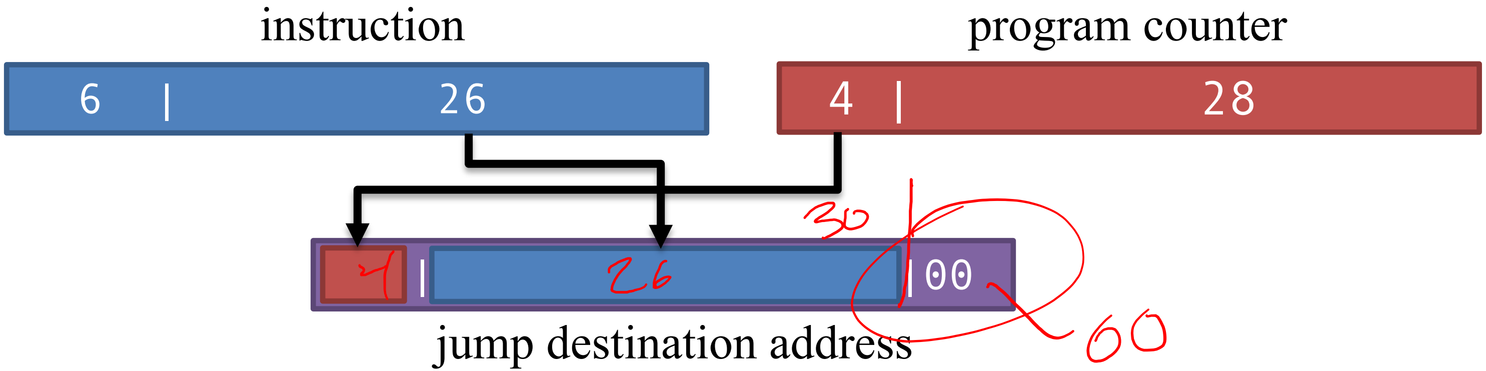
   - Real compilers do this sometimes

How do you specify the destination of a branch/jump?

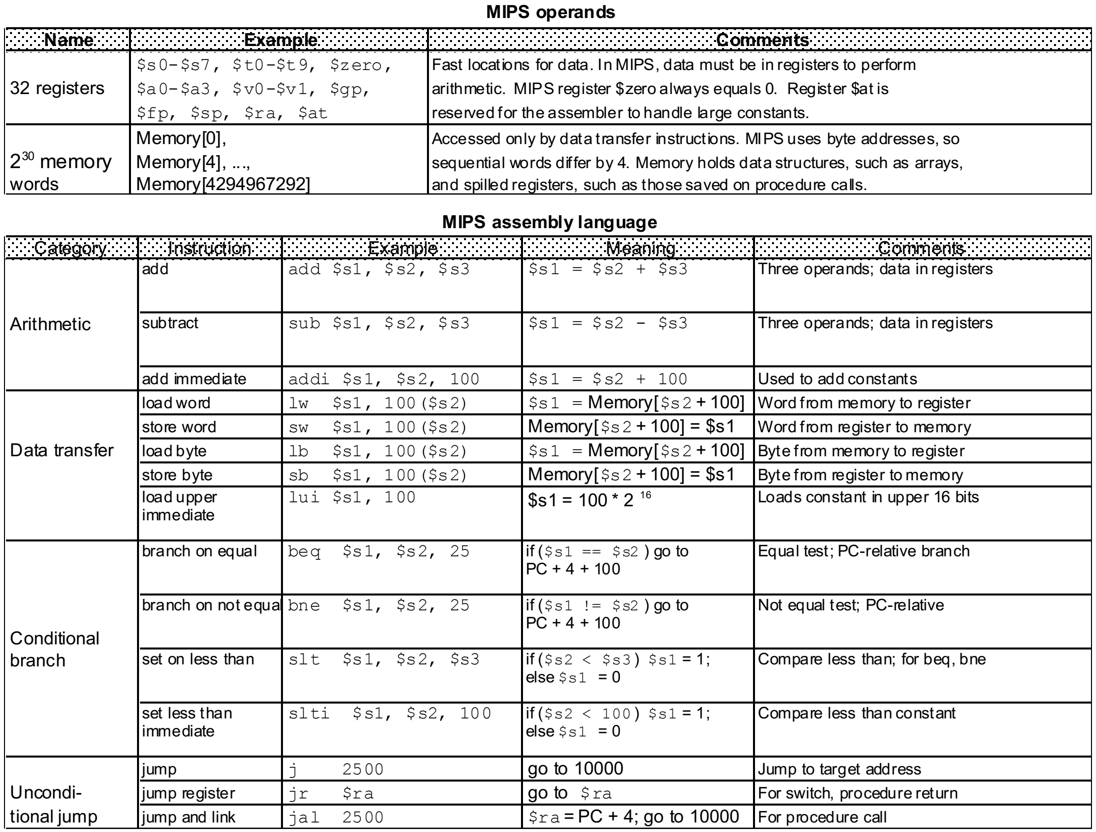
- Unconditional jumps may go long distances
  - Function calls, returns, …
- Studies show that almost all conditional branches go short distances from the current program counter
  - loops, if-then-else, …
- A relative address requires (many) fewer bits than an absolute address
  - e.g., `beq $1, $2, 100  =>  if ($1 == $2):  PC = (PC+4) + 100 * 4`

MIPS Branch and Jump Addressing Modes

- Branches (e.g., beq) use PC-relative addressing mode
  - uses fewer bits since address typically close
  - Aka: base+displacement mode, with the PC being the base
- Jumps use pseudo-direct addressing mode
  - Recall opcode is 6 bits…
    - How many bits are available for displacement? How far can you jump?
  - 26 bits of the address is in the instruction, the rest is taken from the PC.



MIPS in one slide



MIPS operands

Name	Example	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

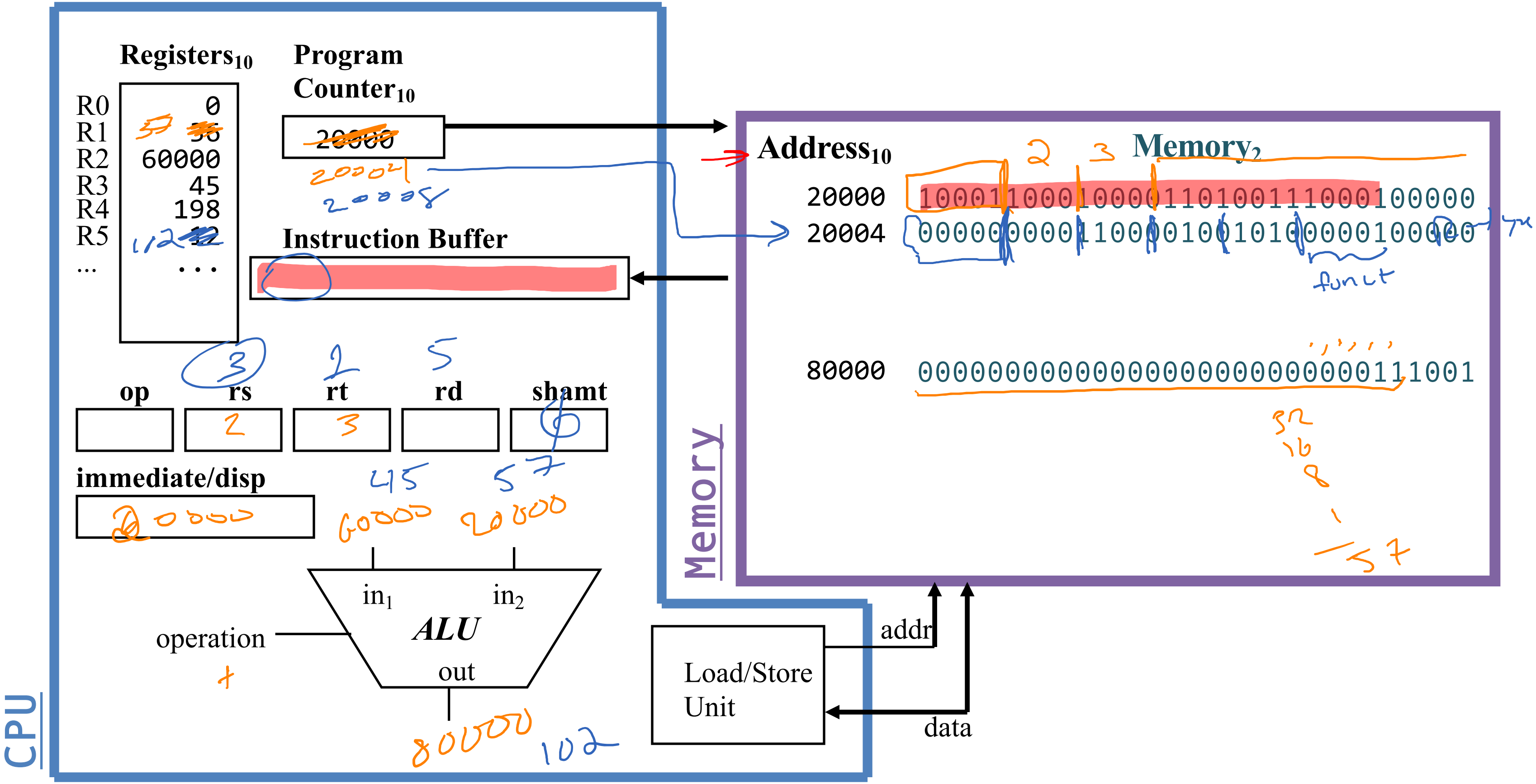
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1, $s2, $s3	$s1 = $s2 + $s3	Three operands; data in registers
	subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3	Three operands; data in registers
	add immediate	addi $s1, $s2, 100	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	lw $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	sw $s1, 100($s2)	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	lb $s1, 100($s2)	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	sb $s1, 100($s2)	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	lui $s1, 100	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq $s1, $s2, 25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equa	bne $s1, $s2, 25	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1, $s2, $s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set less than immediate	slti $s1, $s2, 100	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

Review — Instruction Execution in a CPU

100_011

lw

add



Poll Q: Work an Example

v[k]

$4 $5

v + k * 4

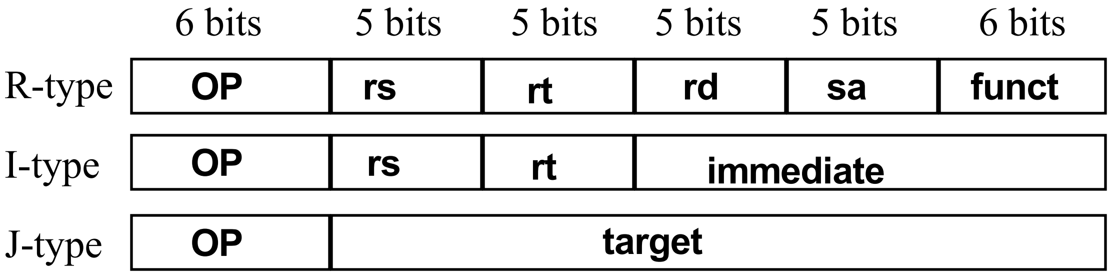
- Can we figure out the code?

```
void
swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
  muli     $2, $5, 4
  add      $2, $4, $2
  lw     $15, 0($2)
  lw     $16, 4($2)
  sw     $16, 0($2)
  sw     $15, 4($2)
  jr     $31
```

	Where is k?
A	$4
B	$5
C	$15
D	$16
E	None of the above

MIPS ISA Tradeoffs



	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R-type	**OP**	**rs**	**rt**	**rd**	**sa**	**funct**
I-type	**OP**	**rs**	**rt**	**immediate**		
J-type	**OP**	**target**				

What if?

- 64 registers
- 20-bit immediates
- 4 operand instruction (e.g. Y = AX + B)

RISC Architectures

- MIPS, like SPARC, PowerPC, and Alpha AXP, is a RISC (Reduced Instruction Set Computer) ISA.
  - fixed instruction length
  - few instruction formats
  - load/store architecture
- RISC architectures worked because they enabled pipelining. They continue to thrive because they enable parallelism.

Alternative Architectures

- Design alternative:
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI (cycles per instruction)
- Sometimes referred to as "RISC vs. CISC"
  - CISC = Complex Instruction Set Computer (as alt to RISC)
  - virtually all new instruction sets since 1982 have been RISC
  - VAX: minimize code size, make assembly language easy
    instructions from 1 to 54 bytes long!
- We'll look (briefly!) at PowerPC and 80x86
- What is ARM?

PowerPC

- Indexed addressing
  - example: `lw $t1,$a0+$s3    # $t1=Memory[$a0+$s3]`
  - What do we have to do in MIPS?

- Update addressing
  - update a register as part of load (for marching through arrays)
  - example: `lwu $t0,4($s3)   # $t0=Memory[$s3+4];$s3=$s3+4`
  - What do we have to do in MIPS?

- Others:
  - load multiple/store multiple
  - a special counter register "`bc Loop`"

    decrement counter, if not 0 goto loop

80x86

1978: The Intel 8086 is announced (16 bit architecture)

1980: The 8087 floating point coprocessor is added

1982: The 80286 increases address space to 24 bits, +instructions

1985: The 80386 extends to 32 bits, new addressing modes

1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)

1997: MMX is added

1999: Pentium III (same architecture)

2001: Pentium 4 (144 new multimedia instructions), simultaneous multithreading (hyperthreading)

2005: dual core Pentium processors

2006: quad core (sort of) Pentium processors

2009: Nehalem – eight-core multithreaded processors

2015: Skylake – 4-core, multithreaded, added hw security features, transactional memory…

80x86

- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes

    e.g., "base or scaled index with 8 or 32 bit displacement"
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

Key Points

- MIPS is a general-purpose register, load-store, fixed-instruction-length architecture.
- MIPS is optimized for fast pipelined performance, not for low instruction count
- Historic architectures favored code size over parallelism.
- MIPS most complex addressing mode, for both branches and loads/stores is base + displacement.