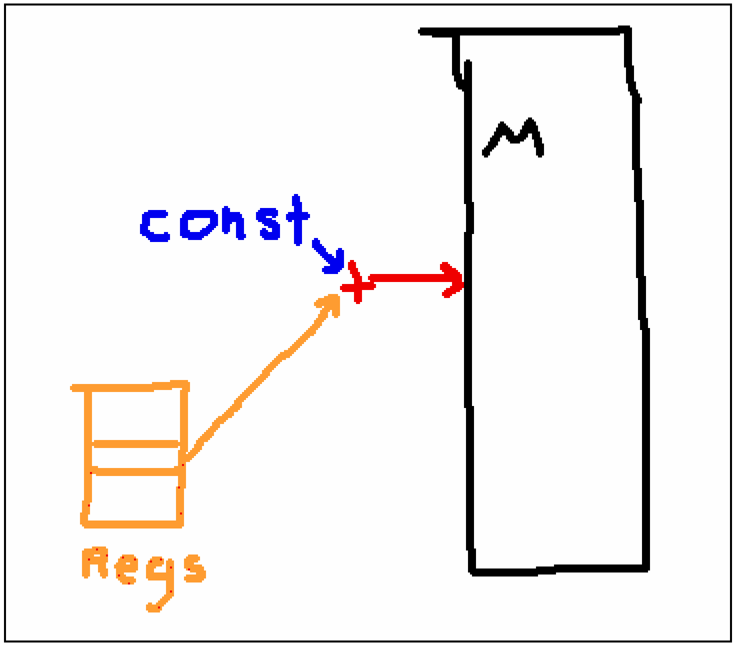
inst.eecs.berkeley.edu/~cs61c/su05

CS61C : Machine Structures

Lecture #7: MIPS Memory & Decisions



(no, *I* didn't draw this…)

2005-06-29

Andy Carle

Review

- In MIPS Assembly Language:
  - Registers replace C variables
  - One Instruction (simple operation) per line
  - Simpler is Better, Smaller is Faster
- New Instructions:

  `add, addi, sub`
- New Registers:

  C Variables: `$s0` - `$s7`

  Temporary Variables: `$t0` - `$t7`

  Zero: `$zero`

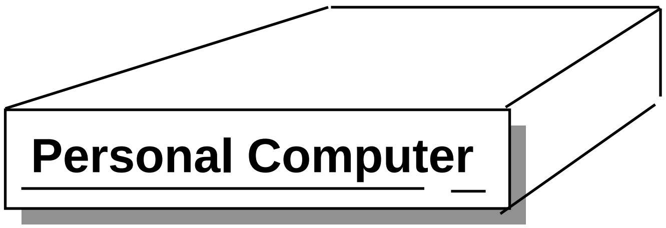
Topic Outline

- **Memory Operations**
- Decisions
- More Instructions

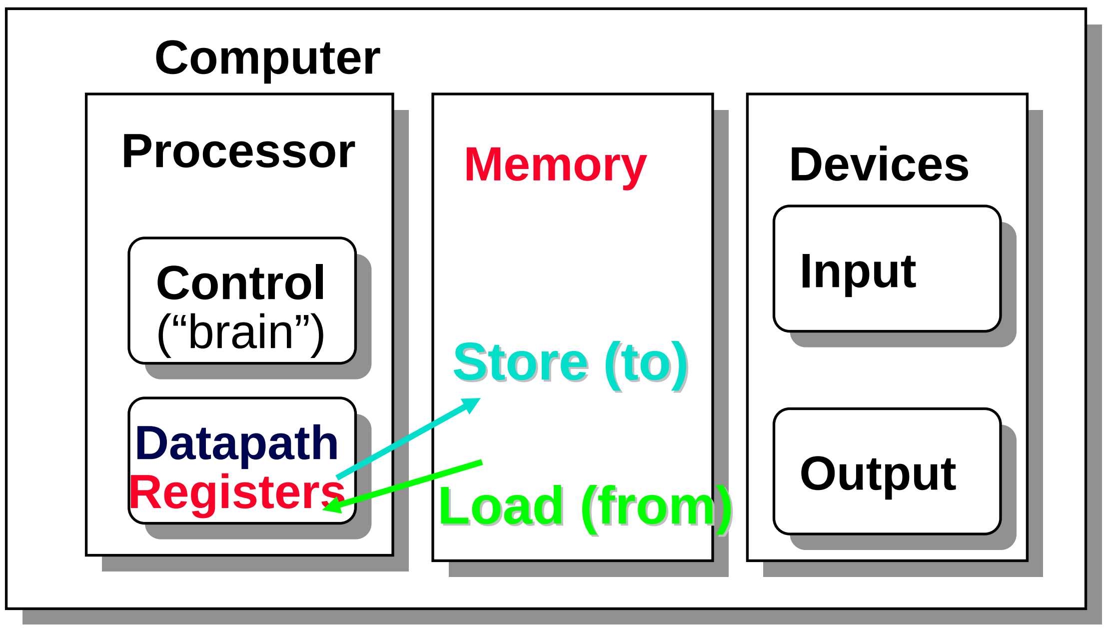
Assembly Operands: Memory

- C variables map onto registers; what about large data structures like arrays?
- 1 of 5 components of a computer: memory contains such data structures
- But MIPS arithmetic instructions only operate on registers, never directly on memory.
- Data transfer instructions transfer data between registers and memory:
    - Memory to register
    - Register to memory

Anatomy: 5 components of any Computer



Registers are in the datapath of the processor; if operands are in memory, we must transfer them to the processor to operate on them, and then transfer back to memory when done.



These are "data transfer" instructions…

Data Transfer: Memory to Reg (1/5)

- To specify a memory address to copy from, specify two things:
  - A register containing a pointer to memory
  - A numerical offset (in bytes)
- The desired memory address is the sum of these two values.
- Example: `8($t0)`
  - specifies the memory address pointed to by the value in `$t0`, plus 8 bytes

Data Transfer: Memory to Reg (2/5)

- **Load Instruction Syntax:**

  lw <reg1> <offset>(<reg2>)

  - **where**

    lw: op name to load a word from memory

    reg1: register that will receive value

    offset: numerical address offset in bytes

    reg2: register containing pointer to memory

Equivalent to:

reg1 ← Memory [reg2 + offset]

Data Transfer: Memory to Reg (3/5)

Example: `lw $t0,12($s0)`

This instruction will take the pointer in `$s0`, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register `$t0`

- **Notes:**
  - `$s0` is called the <u>base register</u>
  - 12 is called the <u>offset</u>
  - offset is generally used in accessing elements of array or structure: base reg points to beginning of array or structure

Data Transfer: Reg to Memory (4/5)

- **Also want to store from register into memory**
  - **Store instruction syntax is identical to Load's**
- **MIPS Instruction Name:**

  `sw` (meaning Store Word, so 32 bits or one word are loaded at a time)

- **Example: `sw $t0,12($s0)`**

  This instruction will take the pointer in `$s0`, add 12 bytes to it, and then store the value from register `$t0` into that memory address

- **Remember: "Store INTO memory"**

Data Transfer: Pointers v. Values (5/5)

- **Key Concept: A register can hold any 32-bit value. That value can be a (signed) `int`, an `unsigned int`, a pointer (memory address), and so on**
- **If you write `lw $t2,0($t0)` then `$t0` better contain a pointer**
- **Don't mix these up!**

Addressing: What's a Word? (1/5)

- A word is the basic unit of the computer.
  - Usually sizeof(word) == sizeof(registers)
  - Can be 32 bits, 64 bits, 8 bits, etc.
  - Not necessarily the smallest unit in the machine!

Addressing: Byte vs. word (2/5)

- **Every word in memory has an address, similar to an index in an array**
- **Early computers numbered words like C numbers elements of an array:**
  - `Memory[0], Memory[1], Memory[2], ...`

  Called the "**address**" of a word

- **Computers needed to access 8-bit bytes as well as words (4 bytes/word)**
- **Today machines address memory as bytes, (i.e.,"Byte Addressed") hence 32-bit (4 byte) word addresses differ by 4**
  - `Memory[0], Memory[4], Memory[8], ...`

Addressing: The Offset Field (3/5)

- What offset in `lw` to select `A[8]` in C?
- 4x8=32 to select `A[8]`: byte v. word
- Compile by hand using registers:

```
g = h + A[8];
```

  - `g`: `$s1`, `h`: `$s2`, `$s3`:base address of `A`

- 1st transfer from memory to register:

```
lw $t0,32($s3)     # $t0 gets A[8]
```

  - Add 32 to `$s3` to select `A[8]`, put into `$t0`

- Next add it to h and place in g

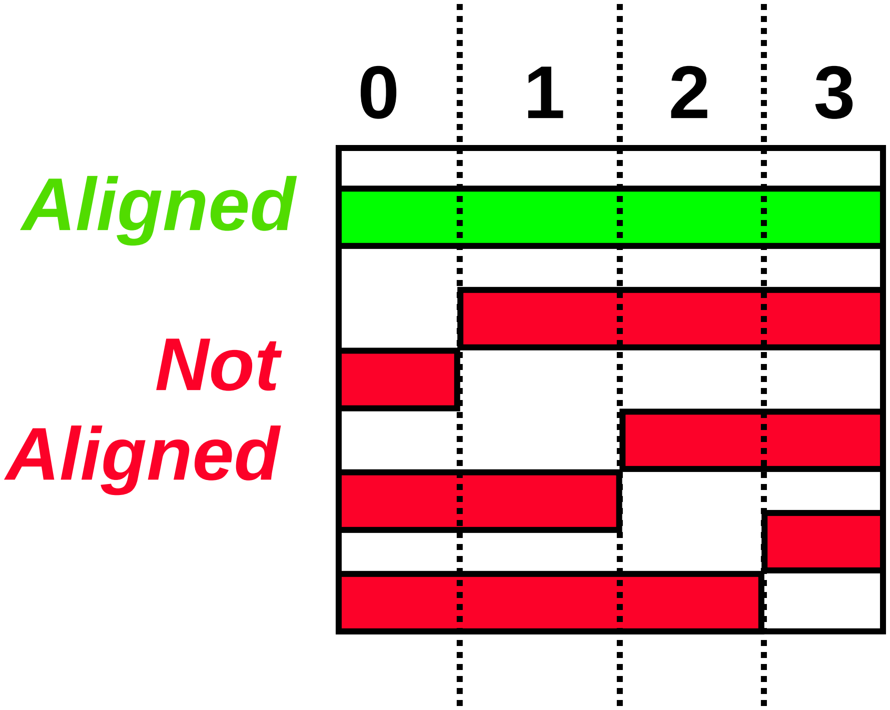
```
add $s1,$s2,$t0   # $s1 = h+A[8]
```

Addressing: Pitfalls (4/5)

- **Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.**
  - **Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes.**
  - **So remember that for both `lw` and `sw`, the sum of the base address and the offset must be a multiple of 4 (to be word aligned)**

Addressing: Memory Alignment (5/5)

- **MIPS requires that all words start at byte addresses that are multiples of 4 bytes**



Last hex digit of address is:

0, 4, 8, or C_{hex}

1, 5, 9, or D_{hex}

2, 6, A, or E_{hex}

3, 7, B, or F_{hex}

- **Called <u>Alignment</u>: objects must fall on address that is multiple of their size.**

Role of Registers vs. Memory

- What if more variables than registers?
  - Compiler tries to keep most frequently used variable in registers
  - Less common in memory: spilling
- Why not keep all variables in memory?
  - registers are faster than memory
- Why not have arithmetic insts to operate on memory addresses?
  - E.g. “addmem 0($s1) 0($s2) 0($s3)”
  - Some ISAs do things like this (x86)
  - MIPS – Keep the common case fast.

Peer Instruction Round 1

We want to translate `*x = *y` into MIPS
(`x`, `y` are pointers stored in: `$s0` `$s1`)

Topic Outline

- Memory Operations
- Decisions
- More Instructions

So Far...

- All instructions so far only manipulate data…we've built a calculator.
- In order to build a computer, we need ability to make decisions…
- C (and MIPS) provide <u>labels</u> to support "`goto`" jumps to places in code.
    - C: Horrible style; MIPS: Necessary!
    - Speed over ease-of-use (again!)

Decisions: C `if` Statements (1/3)

- 2 kinds of `if` statements in C
  - `if (`*condition*`)` *clause*
  - `if (`*condition*`)` *clause1* `else` *clause2*
- Rearrange 2nd `if` into following:

```
if   (condition) goto L1;
      clause2;
      goto L2;
L1:  clause1;

L2:
```

- Not as elegant as `if-else`, but same meaning

Decisions: MIPS Instructions (2/3)

- **Decision instruction in MIPS:**
  - `beq   register1, register2, L1`
  - `beq` is "Branch if (registers are) equal"
    Same meaning as (using C):
    `if  (register1==register2) goto L1`
- **Complementary MIPS decision instruction**
  - `bne   register1, register2, L1`
  - `bne` is "Branch if (registers are) not equal"
    Same meaning as (using C):
    `if  (register1!=register2) goto L1`
- **Called conditional branches**

Decisions: MIPS Goto Instruction (3/3)

- **In addition to conditional branches, MIPS has an unconditional branch:**

```
j    label
```

- **Called a Jump Instruction: jump (or branch) directly to the given label without needing to satisfy any condition**

- **Same meaning as (using C):**
  `goto label`

- **Technically, it's the same* as:**

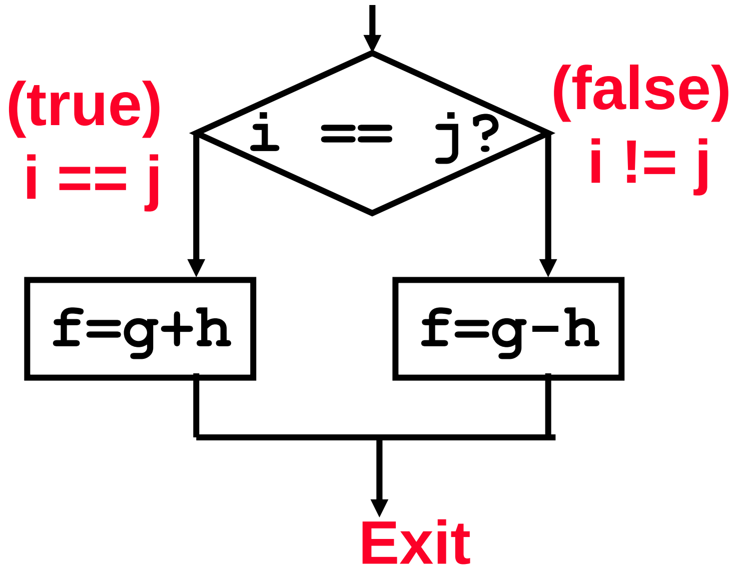
```
beq     $0,$0,label
```

since it always satisfies the condition.

Example: Compiling C `if` into MIPS (1/2)

- Compile by hand

```
if (i == j) f=g+h;
else f=g-h;
```



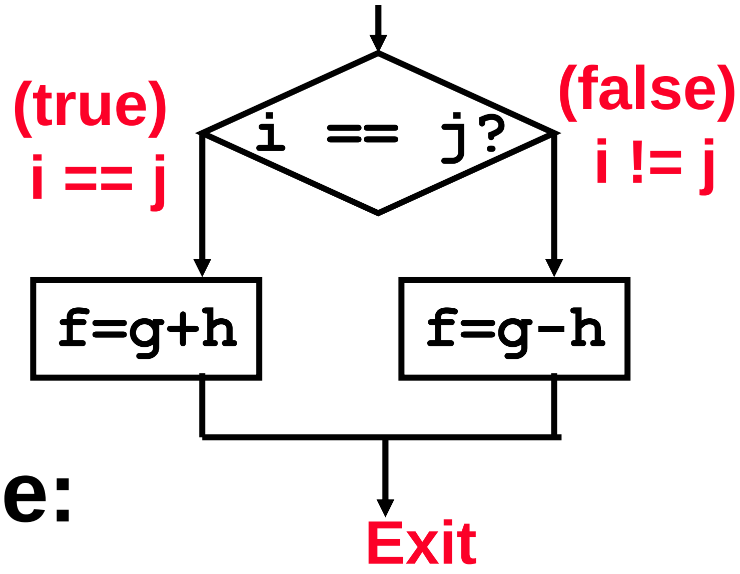
- Use this mapping:

```
f: $s0
g: $s1
h: $s2
i: $s3
j: $s4
```

Example: Compiling C `if` into MIPS (2/2)

- Compile by hand

```
if (i == j) f=g+h;
else f=g-h;
```



- Final compiled MIPS code:

```
      beq $s3,$s4,True   # branch i==j
      sub $s0,$s1,$s2    # f=g-h(false)
      j   Fin            # goto Fin
True: add $s0,$s1,$s2    # f=g+h (true)
Fin:
```

Note: Compiler automatically creates labels to handle decisions (branches). Generally not found in HLL code.

Topic Outline

- **Memory Operations**
- **Decisions**
- **More Instructions**
    - **Memory**
    - **Unsigned**
    - **Logical**
    - **Inequalities**

More Memory Ops: Byte Ops 1/2

- **In addition to word data transfers (`lw`, `sw`), MIPS has byte data transfers:**
  - **load byte: `lb`**
  - **store byte: `sb`**
  - **same format as `lw`, `sw`**

- **`What's the alignment for byte transfers?`**

More Memory Ops: Byte Ops 2/2

- What do with other 24 bits in the 32 bit register?
  - `lb`: sign extends to fill upper 24 bits

`xxxx xxxx xxxx xxxx xxxx xxxx xzzz zzzz`

...is copied to "sign-extend"

byte loaded

This bit

- Normally don't want to sign extend chars
- MIPS instruction that doesn't sign extend when loading bytes:

load byte unsigned: `lbu`

Overflow in Arithmetic (1/2)

- Reminder: Overflow occurs when there is a mistake in arithmetic due to the limited precision in computers.
- Example (4-bit unsigned numbers):

+15	1111
<u>+3</u>	<u>0011</u>
+18	10010

  - But we don't have room for 5-bit solution, so the solution would be 0010, which is +2, and wrong.

Overflow in Arithmetic (2/2)

- **Some languages detect overflow (Ada), some don't (C)**
- **MIPS solution is 2 kinds of arithmetic instructions to recognize 2 choices:**
  - **add (`add`), add immediate (`addi`) and subtract (`sub`) cause overflow to be detected**
  - **add unsigned (`addu`), add immediate unsigned (`addiu`) and subtract unsigned (`subu`) do not cause overflow detection**
- **Compiler selects appropriate arithmetic**
  - **MIPS C compilers produce `addu, addiu, subu`**

Two Logic Instructions (1/1)

- More Arithmetic Instructions
- Shift Left: `sll $s1,$s2,2 #s1=s2<<2`
  - Store in `$s1` the value from `$s2` shifted 2 bits to the left, inserting 0's on right; << in C
  - Before: $0000\ 0002_{hex}$
    $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$
  - After: $0000\ 000\underline{8}_{hex}$
    $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 10\underline{00}_{two}$
  - What arithmetic effect does shift left have?
- Shift Right: `srl` is opposite shift; >>

Inequalities in MIPS (1/3)

- Until now, we've only tested equalities (`==` and `!=` in C). General programs need to test < and > as well.
- Create a MIPS Inequality Instruction:
  - "Set on Less Than"
  - Syntax: `slt  reg1,reg2,reg3`
  - Meaning: `reg1 = (reg2 < reg3);`

```
if (reg2 < reg3)
      reg1 = 1;
else reg1 = 0;
```

  - "set" means "set to 1", "reset" means "set to 0".

Inequalities in MIPS (2/3)

- How do we use this?

```
if (g < h) goto Less; #g:$s0, h:$s1

slt $t0,$s0,$s1 # $t0 = 1 if g<h
bne $t0,$0,Less # goto Less
                # if $t0!=0
                # (if (g<h)) Less:
```

- Branch if `$t0` != 0 → (g < h)
- Register `$0` always contains the value `0`, so `bne` and `beq` often use it for comparison after an `slt` instruction.

Inequalities in MIPS (3/3)

- Now, we can implement <, but how do we implement >, ≤ and ≥ ?
- We could add 3 more instructions, but:
  - MIPS goal: Simpler is Better
- Can we implement ≤ in one or more instructions using just `slt` and the branches?
- What about >?
- What about ≥?

Immediates in Inequalities (1/1)

- There is also an immediate version of `slt` to test against constants: `slti`
  - Helpful in `for` loops

C

```
if (g >= 1) goto Loop
```

MIPS

```
Loop: . . .

slti $t0,$s0,1      # $t0 = 1 if
                    # $s0<1 (g<1)
beq  $t0,$0,Loop    # goto Loop
                    # if $t0==0
                    # (if (g>=1))
```

What about unsigned numbers?

- Also unsigned inequality instructions:

  `sltu`, `sltiu`

…which set result to 1 or 0 depending on unsigned comparisons

- What is value of `$t0`, `$t1`?

(`$s0 = FFFF FFFA`$_{hex}$, `$s1 = 0000 FFFA`$_{hex}$)

```
slt $t0, $s0, $s1
sltu $t1, $s0, $s1
```

MIPS Signed vs. Unsigned – diff meanings!

- **MIPS Signed v. Unsigned is an "overloaded" term**
  - **Do/Don't sign extend (lb, lbu)**
  - **Don't overflow (but still 2s-comp) (addu, addiu, subu, multu, divu)**
  - **Do signed/unsigned compare (slt,slti/sltu,sltiu)**

Loops in C/Assembly (1/3)

- Simple loop in C; `A[]` is an array of `int`s

```
do {
        g = g + A[i];
        i = i + j;
  } while (i != h);
```

- Rewrite this as:

```
Loop: g = g + A[i];
      i = i + j;
      if (i != h) goto Loop;
```

- Use this mapping:

```
   g,   h,   i,   j, base of A
$s1, $s2, $s3, $s4, $s5
```

Loops in C/Assembly (2/3)

- **Final compiled MIPS code:**

```
Loop: sll  $t1,$s3,2     #$t1= 4*I
      add  $t1,$t1,$s5   #$t1=addr A
      lw   $t1,0($t1)    #$t1=A[i]
      add  $s1,$s1,$t1   #g=g+A[i]
      add  $s3,$s3,$s4   #i=i+j
      bne  $s3,$s2,Loop# goto Loop
                         # if i!=h
```

- **Original code:**

```
Loop: g = g + A[i];
      i = i + j;
      if (i != h) goto Loop;
```

Loops in C/Assembly (3/3)

- There are three types of loops in C:
  - `while`
  - `do… while`
  - `for`
- Each can be rewritten as either of the other two, so the method used in the previous example can be applied to `while` and `for` loops as well.
- Key Concept: Though there are multiple ways of writing a loop in MIPS, the key to decision making is conditional branch

Peer Instruction

```
Loop:addi $s0,$s0,-1
     slti $t0,$s1,2
     beq  $t0,$0 ,Loop
     slt  $t0,$s1,$s0
     bne  $t0,$0 ,Loop
```

($s0=i, $s1=j)

What C code properly fills in the blank in loop below?

```
do {i--;} while(__);
```

Summary (1/2)

- Memory is byte-addressable, but `lw` and `sw` access one word at a time.
- A pointer (used by `lw` and `sw`) is just a memory address, so we can add to it or subtract from it (using offset).
- A Decision allows us to decide what to execute at run-time rather than compile-time.
- C Decisions are made using conditional statements within `if`, `while`, `do while`, `for`.
- MIPS Decision making instructions are the conditional branches: `beq` and `bne`.
- New Instructions:

```
lw, sw, beq, bne, j
```

Summary (2/2)

- In order to help the conditional branches make decisions concerning inequalities, we introduce a single instruction: "Set on Less Than"called `slt`, `slti`, `sltu`, `sltiu`
- One can load and store (signed and unsigned) bytes as well as words
- Unsigned add/sub don't detect overflow
- New MIPS Instructions:

```
sll, srl
slt, slti, sltu, sltiu
addu, addiu, subu
```