

`inst.eecs.berkeley.edu/~cs61c`
CS61CL : Machine Structures

Lecture #4 – MIPS I: Registers, Memory, Decisions

2009-06-30



Jeremy Huddleston



Review

- **Data lives in 3 places in memory**
 - **Stack – local variables, function parameters**
 - **Heap – malloc (don't forget to free!)**
 - **Static – global variables**
- **Several techniques for managing heap w/ malloc/free: best-, first-, next-fit, slab, buddy**
 - **2 types of memory fragmentation: internal & external; all suffer from some kind of frag.**
 - **Each technique has strengths and weaknesses, none is definitively best**



Assembly Language

- **Basic job of a CPU: execute lots of *instructions*.**
- **Instructions are the primitive operations that the CPU may execute.**
- **Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an *Instruction Set Architecture (ISA)*.**
 - **Examples: Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS, Intel IA64, ...**



MIPS Architecture

- MIPS – semiconductor company that built one of the first commercial RISC architectures
- We will study the MIPS architecture in some detail in this class (also used in upper division courses CS 152, 162, 164)
- Why MIPS instead of Intel 80x86?
 - MIPS is simple, elegant. Don't want to get bogged down in gritty details.
 - MIPS widely used in embedded apps, x86 little used in embedded, and more embedded computers than PCs



Most HP LaserJet workgroup printers are driven by MIPS-based™ 64-bit processors.



Assembly Variables: Registers (1/4)

- **Unlike HLL like C or Java, assembly cannot use variables**
 - Why not? Keep Hardware Simple
- **Assembly Operands are registers**
 - limited number of special locations built directly into the hardware
 - operations can only be performed on these!
- **Benefit: Since registers are directly in hardware, they are very fast (faster than 1 billionth of a second)**



Assembly Variables: Registers (2/4)

- **Drawback: Since registers are in hardware, there are a predetermined number of them**
 - **Solution: MIPS code must be very carefully put together to efficiently use registers**
- **32 registers in MIPS**
 - **Why 32? *Smaller is faster***
- **Each MIPS register is 32 bits wide**
 - **Groups of 32 bits called a word in MIPS**



Assembly Variables: Registers (3/4)

- **Registers are numbered from 0 to 31**
- **Each register can be referred to by number or name**
- **Number references:**

`$0, $1, $2, ... $30, $31`



Assembly Variables: Registers (4/4)

- **By convention, each register also has a name to make it easier to code**

- **For now:**

\$16 - \$23 → \$s0 - \$s7

(correspond to C variables)

\$8 - \$15 → \$t0 - \$t7

(correspond to temporary variables)

Later will explain other 16 register names

- **In general, use names to make your code more readable**



C, Java variables vs. registers

- In C (and most High Level Languages) variables declared first and given a type

- Example:

```
int fahr, celsius;  
char a, b, c, d, e;
```

- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match int and char variables).
- In Assembly Language, the registers have no type; operation determines how register contents are treated



Comments in Assembly

- **Another way to make your code more readable: comments!**
- **Hash (#) is used for MIPS comments**
 - anything from hash mark to end of line is a comment and will be ignored
 - This is just like the C99 `//`
- **Note: Different from C.**
 - C comments have format
`/* comment */`
so they can span many lines



Assembly Instructions

- In assembly language, each statement (called an **Instruction**), executes exactly one of a short list of simple commands
- Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction
- Instructions are related to operations (=, +, -, *, /) in C or Java
- **Ok, enough already...gimme my MIPS!**



MIPS Addition and Subtraction (1/4)

- **Syntax of Instructions:**

1 2,3,4

where:

1) operation by name

2) operand getting result (“destination”)

3) 1st operand for operation (“source1”)

4) 2nd operand for operation (“source2”)

- **Syntax is rigid:**

- 1 operator, 3 operands

- Why? **Keep Hardware simple via regularity**



Addition and Subtraction of Integers (2/4)

- **Addition in Assembly**

- **Example:** `add $s0, $s1, $s2` (in MIPS)

- Equivalent to: $a = b + c$ (in C)

- where MIPS registers `$s0`, `$s1`, `$s2` are associated with C variables `a`, `b`, `c`

- **Subtraction in Assembly**

- **Example:** `sub $s3, $s4, $s5` (in MIPS)

- Equivalent to: $d = e - f$ (in C)

- where MIPS registers `$s3`, `$s4`, `$s5` are associated with C variables `d`, `e`, `f`



Addition and Subtraction of Integers (3/4)

- How do the following C statement?

`a = b + c + d - e;`

- Break into multiple instructions

```
add $t0, $s1, $s2 # temp = b + c
```

```
add $t0, $t0, $s3 # temp = temp + d
```

```
sub $s0, $t0, $s4 # a = temp - e
```

- Notice: A single line of C may break up into several lines of MIPS.

- Notice: Everything after the hash mark on each line is ignored (comments)



Addition and Subtraction of Integers (4/4)

- How do we do this?

$$f = (g + h) - (i + j);$$

- Use intermediate temporary register

```
add $t0, $s1, $s2      # temp = g + h
add $t1, $s3, $s4      # temp = i + j
sub $s0, $t0, $t1      # f = (g+h) - (i+j)
```



Register Zero

- One particular immediate, the number zero (0), appears very often in code.
- So we define register zero (`$0` or `$zero`) to always have the value 0; eg

`add $s0, $s1, $zero` (in MIPS)

`f = g` (in C)

where MIPS registers `$s0, $s1` are associated with C variables `f, g`

- defined in hardware, so an instruction

`add $zero, $zero, $s0`

 **will not do anything!**

Immediates

- **Immediates are numerical constants.**
- **They appear often in code, so there are special instructions for them.**

- **Add Immediate:**

`addi $s0,$s1,10` (in MIPS)

`f = g + 10` (in C)

where MIPS registers `$s0`, `$s1` are associated with C variables `f`, `g`

- **Syntax similar to add instruction, except that last argument is a number instead of a register.**



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



Data Transfer: Memory to Reg (1/4)

- To transfer a word of data, we need to specify two things:
 - **Register**: specify this by # (\$0 - \$31) or symbolic name (\$s0, ..., \$t0, ...)
 - **Memory address**: more difficult
 - Think of memory as a single one-dimensional array, so we can address it simply by supplying a pointer to a memory address.
 - Other times, we want to be able to **offset** from this pointer.



• Remember: “**Load FROM memory**”

Data Transfer: Memory to Reg (2/4)

- 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 (3/4)

- **Load Instruction Syntax:**

1 2, 3 (4)

- where

1) operation name

2) register that will receive value

3) numerical offset **in bytes**

4) register containing pointer to memory

- **MIPS Instruction Name:**

- **lw** (meaning Load Word, so 32 bits or one word are loaded at a time)



Data Transfer: Memory to Reg (4/4)



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 base register
- `12` is called the offset
- offset is generally used in accessing elements of array or structure: base reg points to beginning of array or structure (note offset must be a **constant known at assembly time**)



Data Transfer: Reg to Memory

- 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 is stored 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”



Pointers v. Values

- **Key Concept:** A register can hold any 32-bit value. That value can be a `char`, an `int`, a pointer (memory addr), and so on
 - E.g., If you write: `add $t2, $t1, $t0` then `$t0` and `$t1` better contain values that can be added
 - E.g., If you write: `lw $t2, 0($t0)` then `$t0` better contain a pointer
- Don't mix these up!



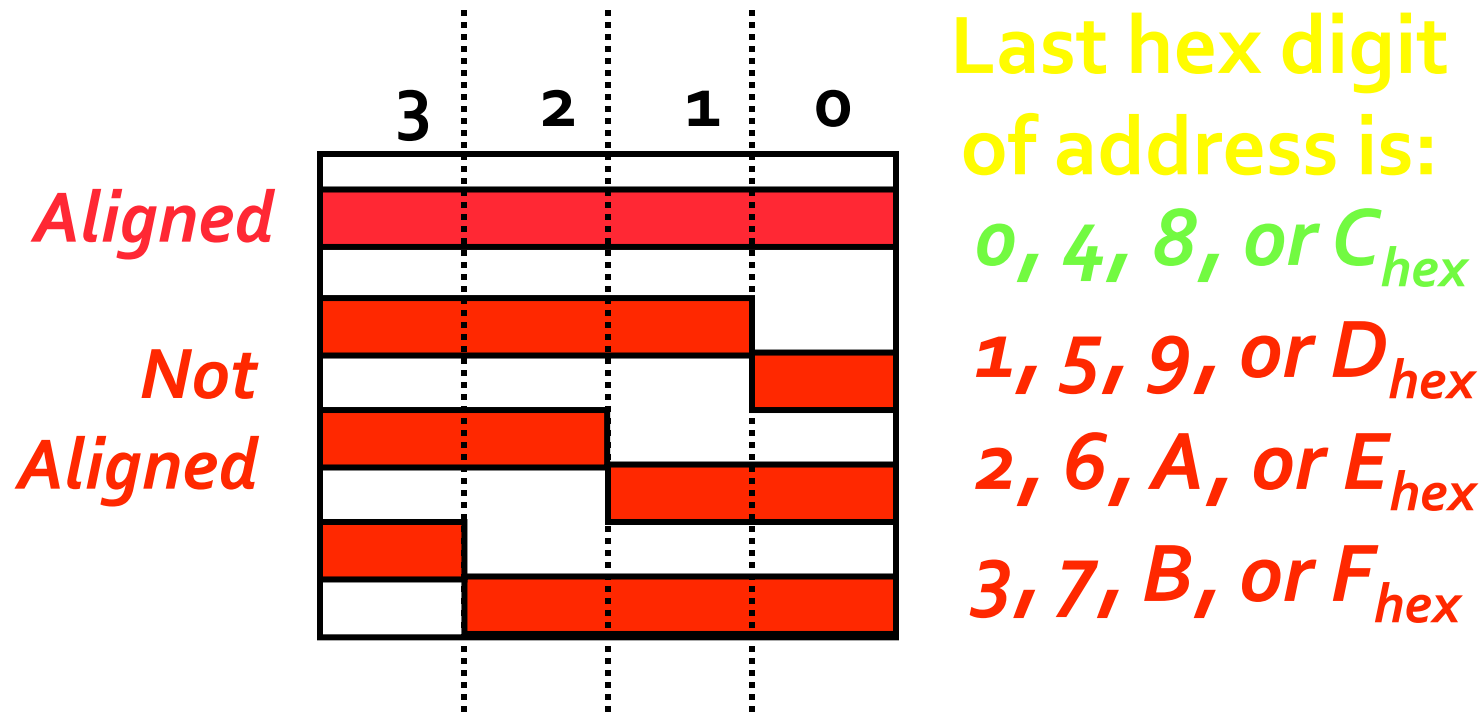
Notes about Memory

- **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.**
 - **Also, 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**)**



More Notes about Memory: Alignment

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



- Called Alignment: objects 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 variables in memory:**
[spilling](#)
- **Why not keep all variables in memory?**
 - **Smaller is faster:**
registers are faster than memory
 - **Registers more versatile:**
 - **MIPS arithmetic instructions can read 2, operate on them, and write 1 per instruction**
 - **MIPS data transfer only read or write 1 operand per instruction, and no operation**



Administrivia

- **HW2 due tomorrow.**
- **HW3 is up.**
- **Proj1 will be up soon... start early**
- **Future “Wednesday” assignments will be moved to Thursday due dates.**
- **Check the newsgroup often and ask there for help.**



So Far...

- All instructions so far only manipulate data...we've built a **calculator** of sorts.
- In order to build a **computer**, we need ability to make decisions...
- C (and MIPS) provide labels to support “**goto**” jumps to places in code.
 - C: Horrible style; **MIPS: Necessary!**



MIPS Decision Instructions

- 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



MIPS Goto Instruction

- 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 effect as:

`beq $0, $0, label`



since it always satisfies the condition.

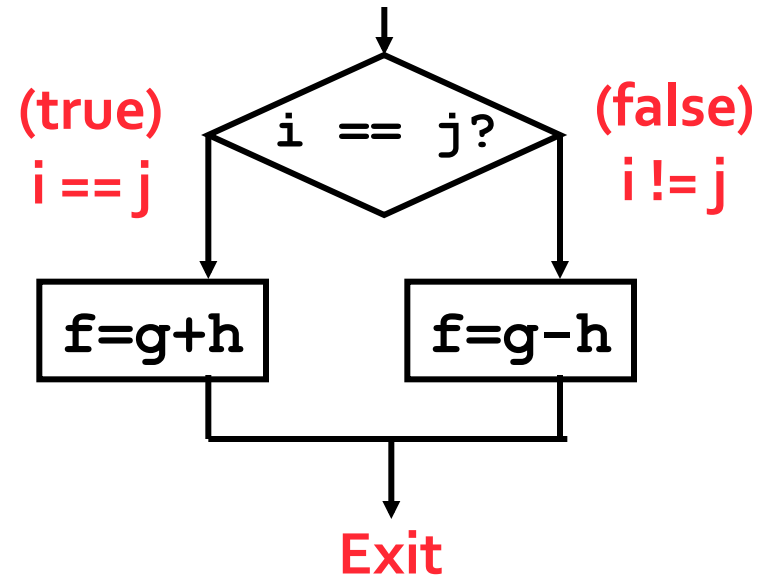
Compiling C `if` into MIPS (1/2)

- 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
```



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

- Compile by hand

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

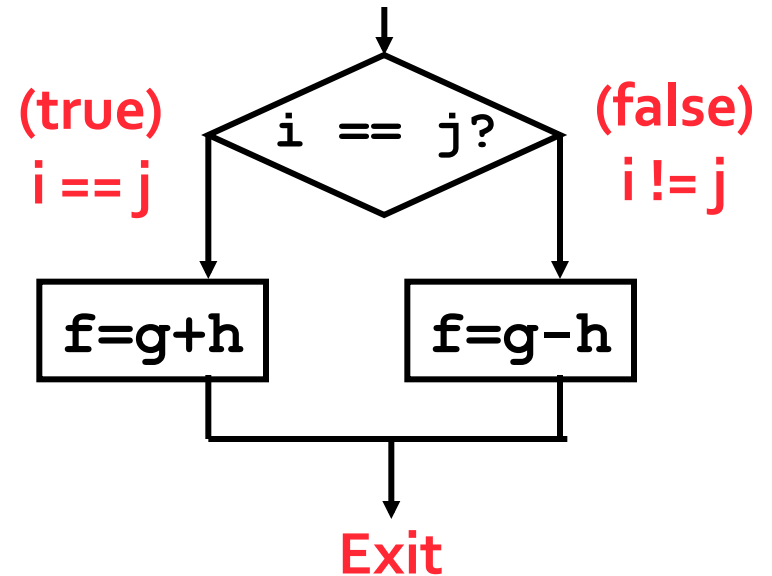
- Use this mapping:

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

- 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.



Loops in C/Assembly (1/3)

- Simple loop in C; **A[]** is an array of `ints`

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

- Rewrite this as:

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

- Use this mapping:

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



Loops in C/Assembly (2/3)

- Final compiled MIPS code:

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

- Original code:

```
Loop:      g = g + A[i];
          i = i + j;
          if (i != ) 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 these loops as well.
- **Key Concept:** Though there are multiple ways of writing a loop in MIPS, the key to decision-making is conditional branch



Inequalities in MIPS (1/4)

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

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

Same thing...

“set” means “change to 1”,
“reset” means “change to 0”.



Inequalities in MIPS (2/4)

- How do we use this? Compile by hand:
`if (g < h) goto Less; #g:$s0, h:$s1`

- Answer: compiled MIPS code...

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

- Register \$0 always contains the value 0, so `bne` and `beq` often use it for comparison after an `slt` instruction.

- A `slt` → `bne` pair means `if (... < ...) goto...`



Inequalities in MIPS (3/4)

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



Inequalities in MIPS (4/4)

```
# a:$s0, b:$s1
slt $t0,$s0,$s1 # $t0 = 1 if a<b
beq $t0,$0,skip # skip if a >= b
    <stuff>      # do if a<b

skip:
```

Two independent variations possible:

Use `slt $t0,$s1,$s0` instead of

`slt $t0,$s0,$s1`

Use `bne` instead of `beq`



Immediates in Inequalities

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

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

M `Loop: . . .`
I `slti $t0,$s0,1` *# \$t0 = 1 if*
P `beq $t0,$0,Loop` *# \$s0 < 1 (g < 1)*
S *# goto Loop*
 # if \$t0 == 0
 # (if (g >= 1))

 An `slt` → `beq` pair means `if (... ≥ ...) goto...`

“And in Conclusion...”

- **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 - \$t9
 - Zero: \$zero



“And in Conclusion...”

- 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, 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`

“And in conclusion...”

- To help the **conditional branches** make decisions concerning inequalities, we introduce: “Set on Less Than” called **slt, slti, sltu, sltiu**
- One can store and load (signed and unsigned) **bytes** as well as words with **lb, lbu**
- Unsigned add/sub **don't cause overflow**
- New MIPS Instructions:



Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

Bonus



Example: The C Switch Statement (1/3)

- Choose among four alternatives depending on whether `k` has the value 0, 1, 2 or 3.
Compile this C code:

```
switch (k) {  
    case 0: f=i+j; break; /* k=0 */  
    case 1: f=g+h; break; /* k=1 */  
    case 2: f=g-h; break; /* k=2 */  
    case 3: f=i-j; break; /* k=3 */  
}
```



Example: The C Switch Statement (2/3)

- This is complicated, so **simplify**.
- Rewrite it as a chain of if-else statements, which we already know how to compile:

```
if (k==0) f=i+j;
    else if (k==1) f=g+h;
        else if (k==2) f=g-h;
            else if (k==3) f=i-j;
```

- Use this mapping:

```
f: $s0, g: $s1, h: $s2,
i: $s3, j: $s4, k: $s5
```



Example: The C Switch Statement (3/3)

- Final compiled MIPS code:

```
        bne $s5,$0,L1      # branch k!=0
        add $s0,$s3,$s4    #k==0 so f=i+j
        j   Exit          # end of case so Exit
L1:     addi $t0,$s5,-1     # $t0=k-1
        bne $t0,$0,L2     # branch k!=1
        add $s0,$s1,$s2    #k==1 so f=g+h
        j   Exit          # end of case so Exit
L2:     addi $t0,$s5,-2     # $t0=k-2
        bne $t0,$0,L3     # branch k!=2
        sub $s0,$s1,$s2    #k==2 so f=g-h
        j   Exit          # end of case so Exit
L3:     addi $t0,$s5,-3     # $t0=k-3
        bne $t0,$0,Exit    # branch k!=3
        sub $s0,$s3,$s4    #k==3 so f=i-j
Exit:
```



Immediates

- There is no Subtract Immediate in MIPS: Why?
- Limit types of operations that can be done to absolute minimum
 - if an operation can be decomposed into a simpler operation, don't include it
 - `addi ..., -X = subi ..., X` \Rightarrow so no `subi`

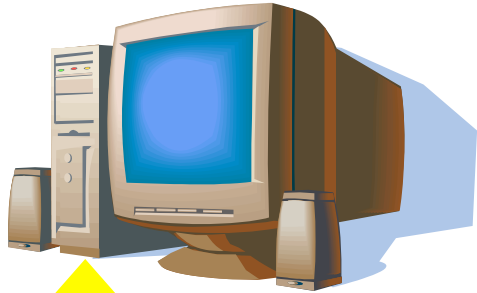
- `addi $s0, $s1, -10` (in MIPS)

$$f = g - 10 \text{ (in C)}$$

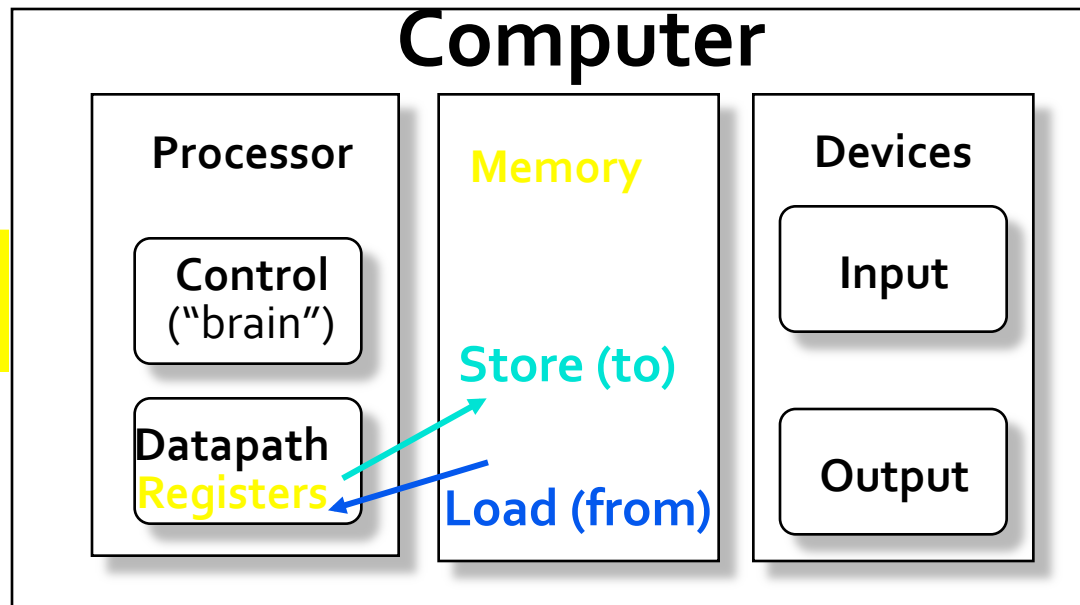
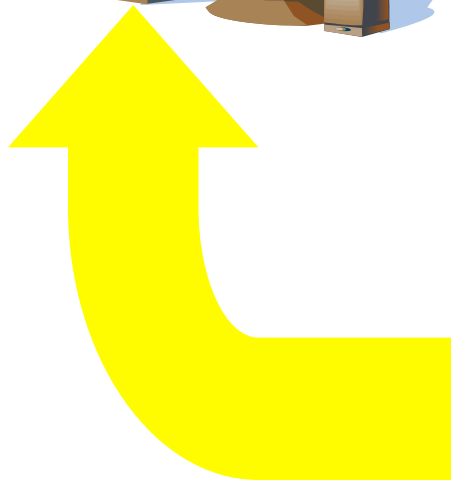
where MIPS registers `$s0`, `$s1` are associated with C variables `f`, `g`



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...



Addressing: Byte vs. Word

- 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]




Compilation with Memory

- What offset in `lw` to select `A[5]` in C?
- $4 \times 5 = 20$ to select `A[5]`: byte v. word
- Compile by hand using registers:
 $g = h + A[5];$
 - `g`: `$s1`, `h`: `$s2`, `$s3`: base address of `A`
- 1st transfer from memory to register:

```
lw      $t0, 20($s3) # $t0 gets  
A[5]
```

- Add 20 to `$s3` to select `A[5]`, put into `$t0`

 • Next add it to `h` and place in `g`

```
add $s1, $s2, $t0 # $s1 = h + A[5]
```

C Decisions: `if` Statements

- 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



Last time: Loading, Storing bytes 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`
- E.g., `lb $s0, 3($s1)`
 - *contents of memory location with address = sum of “3” + contents of register `s1` is copied to the low byte position of register `s0`.*



Loading, Storing bytes 2/2

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



- 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 instructs:
 - These cause overflow to be detected
 - add (**add**)
 - add immediate (**addi**)
 - subtract (**sub**)
 - These do not cause overflow detection
 - add unsigned (**addu**)
 - add immediate unsigned (**addiu**)
 - subtract unsigned (**subu**)



- **Compiler selects appropriate arithmetic**

What about unsigned numbers?

- Also **unsigned** inequality instructions:

`sltu, sltiu`

...which sets result to 1 or 0 depending on unsigned comparisons

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

(`$s0 = FFFF FFFAhex`, `$s1 = 0000 FFFAhex`)

`slt $t0, $s0, $s1`

`sltu $t1, $s0, $s1`



MIPS Signed vs. Unsigned – diff meaning

- MIPS terms Signed/Unsigned “overloaded”:
 - Do/Don't sign extend
 - (lb, lbu)
 - Do/Don't overflow
 - (add, addi, sub, mult, div)
 - (addu, addiu, subu, multu, divu)
 - Do signed/unsigned compare
 - (slt, slti/sltu, sltiu)



Two “Logic” Instructions

- Here are 2 more new 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 0002hex`
`0000 0000 0000 0000 0000 0000 0000 0010two`
 - After: `0000 0008hex`
`0000 0000 0000 0000 0000 0000 0000 1000two`
 - What arithmetic effect does shift left have?
- Shift Right: `srl` is opposite shift; >>

