CS 61C:
Great Ideas in Computer Architecture
Intro to Assembly Language, MIPS Intro

Instructors:
Krste Asanovic & Vladimir Stojanovic
http://inst.eecs.Berkeley.edu/~cs61c/sp15

Levels of Representation/ Interpretation

High Level Language Program (e.g., C)

Compiler

Assembly Language Program (e.g., MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)

Architecture Implementation

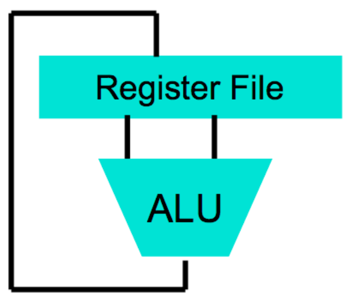
Logic Circuit Description (Circuit Schematic Diagrams)

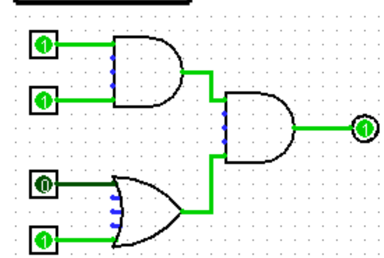
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```





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: ARM, Intel x86, MIPS, RISC-V, IBM/Motorola PowerPC (old Mac), Intel IA64, ...

Instruction Set Architectures

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
  - VAX architecture had an instruction to multiply polynomials!

- RISC philosophy (Cocke IBM, Patterson, Hennessy, 1980s) – Reduced Instruction Set Computing
  - Keep the instruction set small and simple, makes it easier to build fast hardware.
  - Let software do complicated operations by composing simpler ones.

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 x86?
  - 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

Assembly Variables: Registers

- 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 ns - light travels 30cm in 1 ns!!!)

Number of MIPS Registers

- 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, but too small is bad. Goldilocks problem.
- Each MIPS register is 32 bits wide
  - Groups of 32 bits called a word in MIPS

Names of MIPS Registers

- Registers are numbered from 0 to 31
- Each register can be referred to by number or name
- Number references:
  - $0, $1, $2, … $30, $31
- 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, registers have no type; operation determines how register contents are treated

Addition and Subtraction of Integers

- Addition in Assembly
  - Example: `add $s0,$s1,$s2` (in MIPS)
  - Equivalent to: a = b + c (in C)

  where C variables ⇔ MIPS registers are:

  a ⇔ $s0, b ⇔ $s1, c ⇔ $s2
- Subtraction in Assembly
  - Example: `sub $s3,$s4,$s5` (in MIPS)
  - Equivalent to: d = e - f (in C)

  where C variables ⇔ MIPS registers are:

  d ⇔ $s3, e ⇔ $s4, f ⇔ $s5

Addition and Subtraction of Integers Example 1

- How to 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
```

- A single line of C may break up into several lines of MIPS.
- Notice the use of temporary registers – don't want to modify the variable registers `$s`
- Everything after the hash mark on each line is ignored (comments)

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.

```
add $s0,$s1,$zero  (in MIPS)
f = g              (in C)
```

Overflow in Arithmetic

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

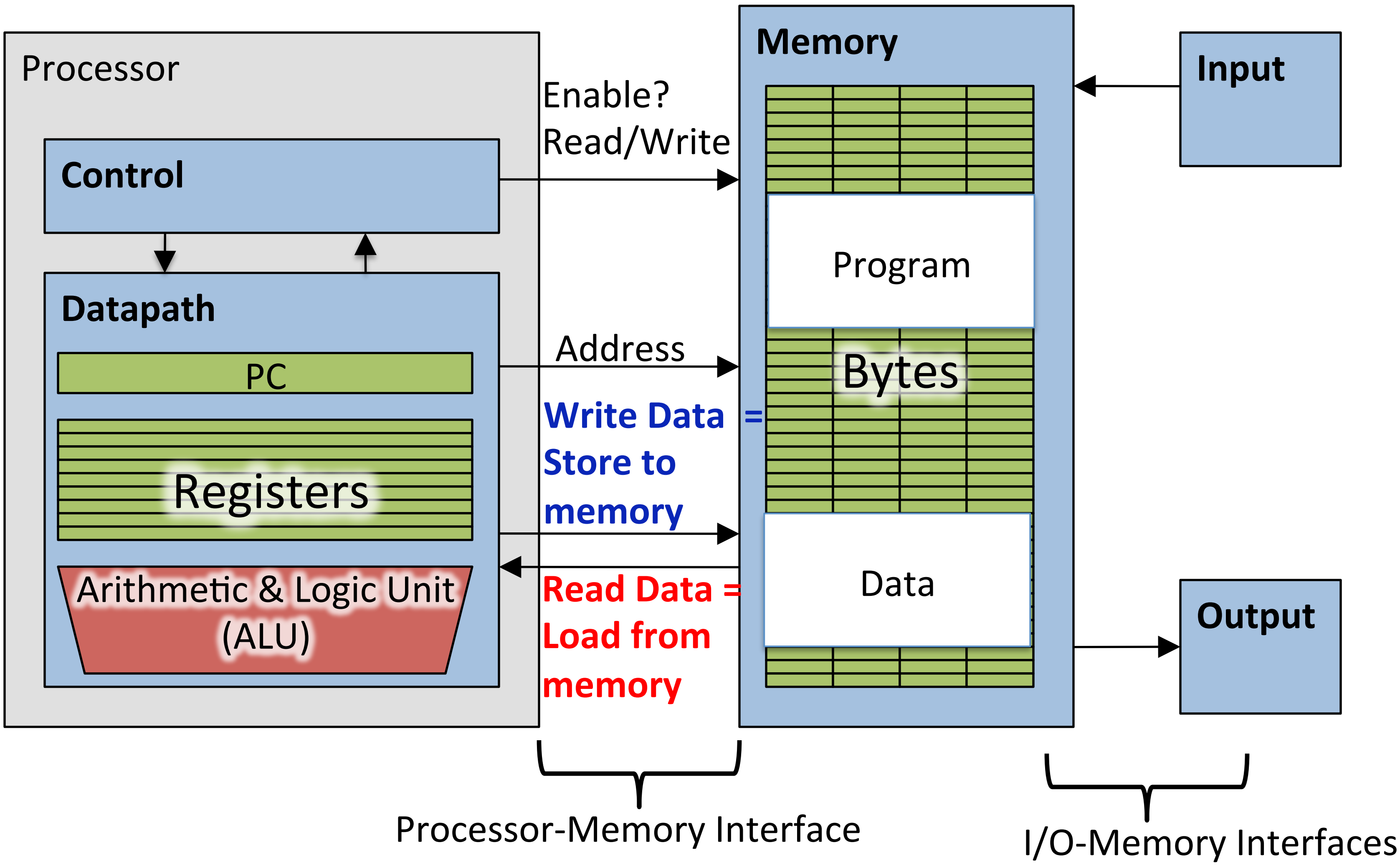
```
   15          1111
+  3        + 0011
   18         10010
```

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

Overflow handling in MIPS

- Some languages detect overflow (Ada), some don't (most C implementations)
- MIPS solution is 2 kinds of arithmetic instructions:
  - 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
  - MIPS C compilers produce addu, addiu, subu

Data Transfer: Load from and Store to memory



Memory Addresses are in Bytes

- Lots of data is smaller than 32 bits, but rarely smaller than 8 bits – works fine if everything is a multiple of 8 bits
- 8 bit chunk is called a *byte* (1 word = 4 bytes)
- Memory addresses are really in *bytes*, not words
- Word addresses are 4 bytes apart
  - Word address is same as address of leftmost byte (i.e. Big-endian)

Addr of lowest byte in word is addr of word

...	...	...	...
12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Transfer from Memory to Register

- C code

```
int  A[100];
g = h + A[3];
```

- Using Load Word (`lw`) in MIPS:

```
lw $t0,12($s3)     # Temp reg $t0 gets A[3]
add $s1,$s2,$t0    # g = h + A[3]
```

Note: `$s3` – base register (pointer)
`12` – offset in bytes
Offset must be a constant known at assembly time

Transfer from Register to Memory

- C code

```
int  A[100];
A[10] = h + A[3];
```

- Using Store Word (sw) in MIPS:

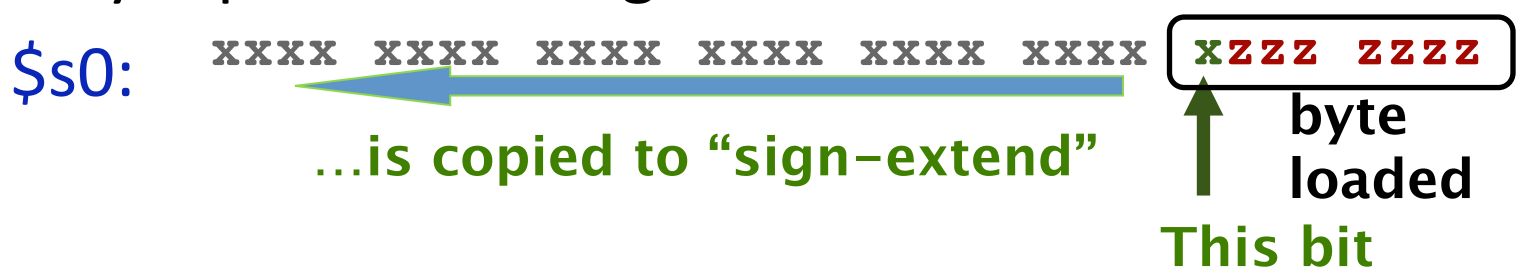
```
lw   $t0,12($s3)    # Temp reg $t0 gets A[3]
add  $t0,$s2,$t0    # Temp reg $t0 gets h + A[3]
sw   $t0, 40($s3)   # A[10] = h + A[3]
```

Note: $s3 – base register (pointer)
12,40 – offsets in bytes

$s3+12 and $s3+40 must be multiples of 4

Loading and Storing bytes

- 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*.



Speed of Registers vs. Memory

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory: Billions of bytes (2 GB to 8 GB on laptop)
- and the RISC principle is…
  - Smaller is faster
- How much faster are registers than memory??
- About 100-500 times faster!
  - in terms of *latency* of one access

How many hours h on Homework 0?

A: $0 \le h < 5$

B: $5 \le h < 10$

C: $10 \le h < 15$

D: $15 \le h < 20$

E: $20 \le h$

Clickers/Peer Instruction

We want to translate `*x = *y +1` into MIPS
(x, y pointers stored in: `$s0 $s1`)

A:
```
addi $s0,$s1,1
```

B:
```
lw   $s0,1($s1)
sw   $s1,0($s0)
```

C:
```
lw   $t0,0($s1)
addi $t0,$t0,1
sw   $t0,0($s0)
```

D:
```
sw   $t0,0($s1)
addi $t0,$t0, 1
lw   $t0,0($s0)
```

E:
```
lw   $s0,1($t0)
sw   $s1,0($t0)
```

MIPS Logical Instructions

- Useful to operate on fields of bits within a word
  - e.g., characters within a word (8 bits)
- Operations to pack /unpack bits into words
- Called *logical operations*

Logical operations	C operators	Java operators	MIPS instructions
Bit-by-bit AND	&	&	**`and`**
Bit-by-bit OR	|	|	**`or`**
Bit-by-bit NOT	~	~	**`not`**
Shift left	<<	<<	**`sll`**
Shift right	>>	>>>	**`srl`**

Logic Shifting

- Shift Left: `sll $s1,$s2,2` #s1=s2<<2
  - Store in $s1 the value from $s2 shifted 2 bits to the left (they fall off end), inserting 0's on right; << in C.

  Before: 0000 0002_{hex}

  $\text{0000 0000 0000 0000 0000 0000 0000 0010}_{two}$

  After: $\text{0000 000}\underline{8}_{hex}$

  $\text{0000 0000 0000 0000 0000 0000 0000 10}\underline{00}_{two}$

What arithmetic effect does shift left have?

- Shift Right: `srl` is opposite shift; >>

Arithmetic Shifting

- Shift right arithmetic moves *n* bits to the right (insert high order sign bit into empty bits)
- For example, if register $s0 contained

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 0111_{two} = -25_{ten}$

- If executed sra $s0, $s0, 4, result is:

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$

- Unfortunately, this is NOT same as dividing by 2^n
  - Fails for odd negative numbers
  - C arithmetic semantics is that division should round towards 0

Computer Decision Making

- Based on computation, do something different
- In programming languages: *if*-statement

- MIPS: *if*-statement instruction is

  `beq register1,register2,L1`

  means: go to statement labeled L1
  if (value in register1) == (value in register2)
  ....otherwise, go to next statement
- `beq` stands for *branch if equal*
- Other instruction: `bne` for *branch if not equal*

Types of Branches

- **Branch** – change of control flow

- **Conditional Branch** – change control flow depending on outcome of comparison
  – branch *if* equal (`beq`) or branch *if not* equal (`bne`)

- **Unconditional Branch** – always branch
  – a MIPS instruction for this*: jump (`j`)*

Example *if* Statement

- Assuming translations below, compile *if* block

  f → `$s0` g → `$s1` h → `$s2`

  i → `$s3` j → `$s4`

```
if (i == j)           bne $s3,$s4,Exit
  f = g + h;          add $s0,$s1,$s2
                Exit:
```

- May need to negate branch condition

Example *if-else* Statement

- Assuming translations below, compile

  f → `$s0` g → `$s1` h → `$s2`

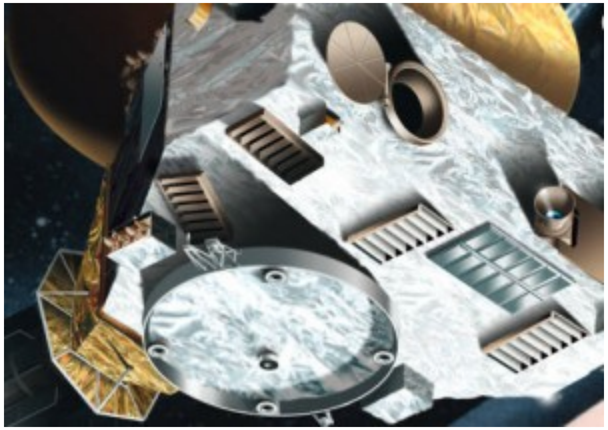
  i → `$s3` j → `$s4`

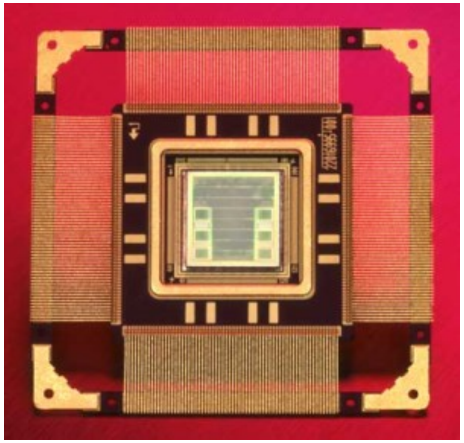
```
if (i == j)                  bne $s3,$s4,Else
  f = g + h;                 add $s0,$s1,$s2
else                         j Exit
  f = g - h;           Else: sub $s0,$s1,$s2
                       Exit:
```

Administrivia

- Hopefully everyone turned-in HW0
- HW1 due 11:59:59pm Sunday 2/8

In the News
MIPS steers spacecraft to Pluto





- 4 MIPS R3000 32bit CPUs
  - Command and Data handling
  - Guidance and Control

- Launched 2006, first pics in July 2015

http://www.electronicsweekly.com/news/military-aerospace-electronics/mips-steers-spacecraft-pluto-2015-01/

Inequalities in MIPS

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

  "set" means "change to 1",

  "reset" means "change to 0".

Inequalities in MIPS Cont.

- 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 $t0,$zero,Less # if $t0!=0 goto Less
```

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

- `sltu` treats registers as unsigned

Immediates in Inequalities

- `slti` an immediate version of `slt` to test against constants

```
Loop:   . . .

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

Loops in C/Assembly

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

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

- Use this mapping: g, h, i, j, &A[0]
  $s1, $s2, $s3, $s4, $s5

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

And In Conclusion …

- Computer words and vocabulary are called instructions and instruction set respectively
- MIPS is example RISC instruction set in this class
- Rigid format: 1 operation, 2 source operands, 1 destination
  - `add,sub,mul,div,and,or,sll,srl,sra`
  - `lw,sw,lb,sb` to move data to/from registers from/to memory
  - `beq, bne, j, slt, slti` for decision/flow control
- Simple mappings from arithmetic expressions, array access, if-then-else in C to MIPS instructions