EE108B
Lecture 2
MIPS Assembly Language I

Christos Kozyrakis
Stanford University
http://eeclass.stanford.edu/ee108b

Announcements

  - EE undergrads: EE108A and CS106B
  - Everybody else: E40 and CS106B (or equivalent)
- EE 108B Problem Session
  - Fridays 2.15pm – 3.05pm, Thornton 102, Live on E4
- HW1 out today
- LAB1 and PA1 out next week
  - Make sure you register for the class
  - Register at eeclass.stanford.edu/ee108b
- Details about lab arrangements to be announced soon
  - Check your email and web page announcements

Review

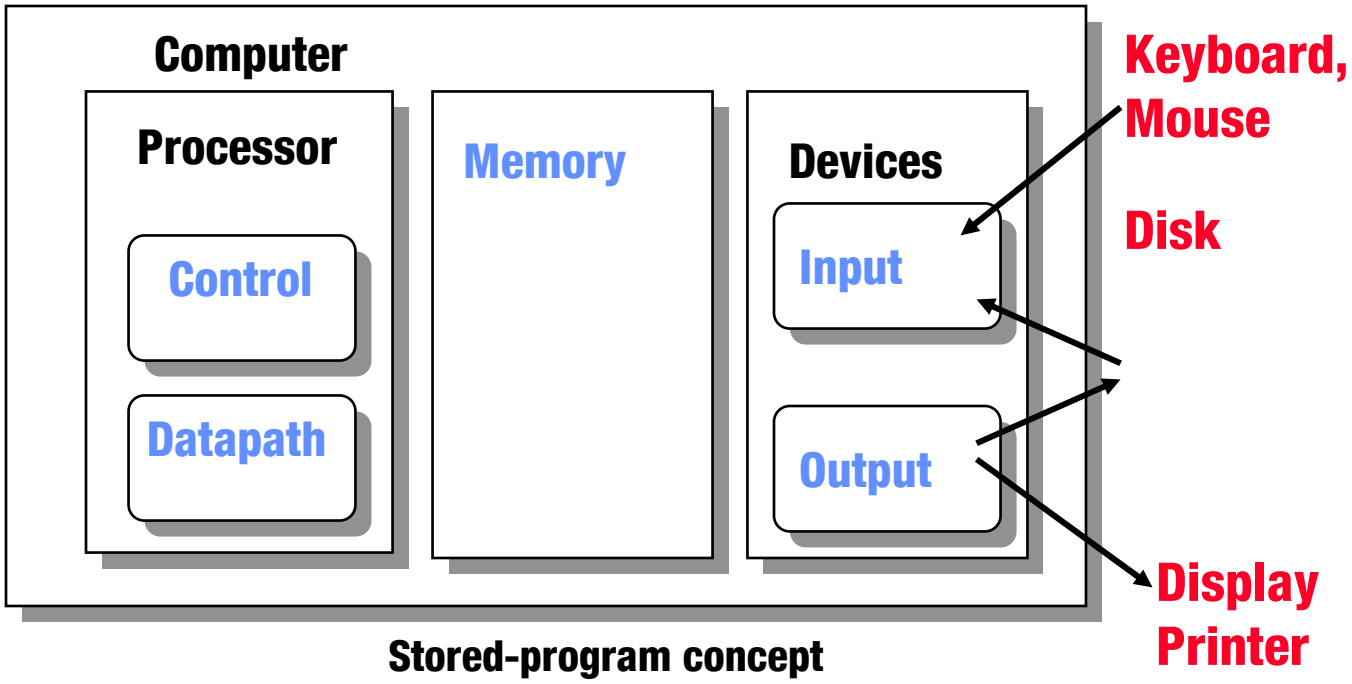
- Integrated circuits (Moore's Law)
  - 2x transistors every 18 months
  - ~300 million transistors today
  - Slowing down due to cost of fabrication plants
- Processors
  - Hide complexity of IC design with sequential programming model
  - Until recently 60% performance improvement per year
  - Slowing sown due to high power, high complexity, and lack of ideas
- Why study computer design?
  - Processors are important
  - Good example of digital system with general design principles
  - Help write better (fewer bugs, higher performance) programs

Today's Topic: MIPS Instruction Set Architecture

- Textbook reading
  - 2.1–2.6
  - Look at how instructions are defined and represented

- What is an instruction set architecture (ISA)?
- Interplay of C and MIPS ISA
- Components of MIPS ISA
  - Register operands
  - Memory operands
  - Arithmetic operations
  - Control flow operations

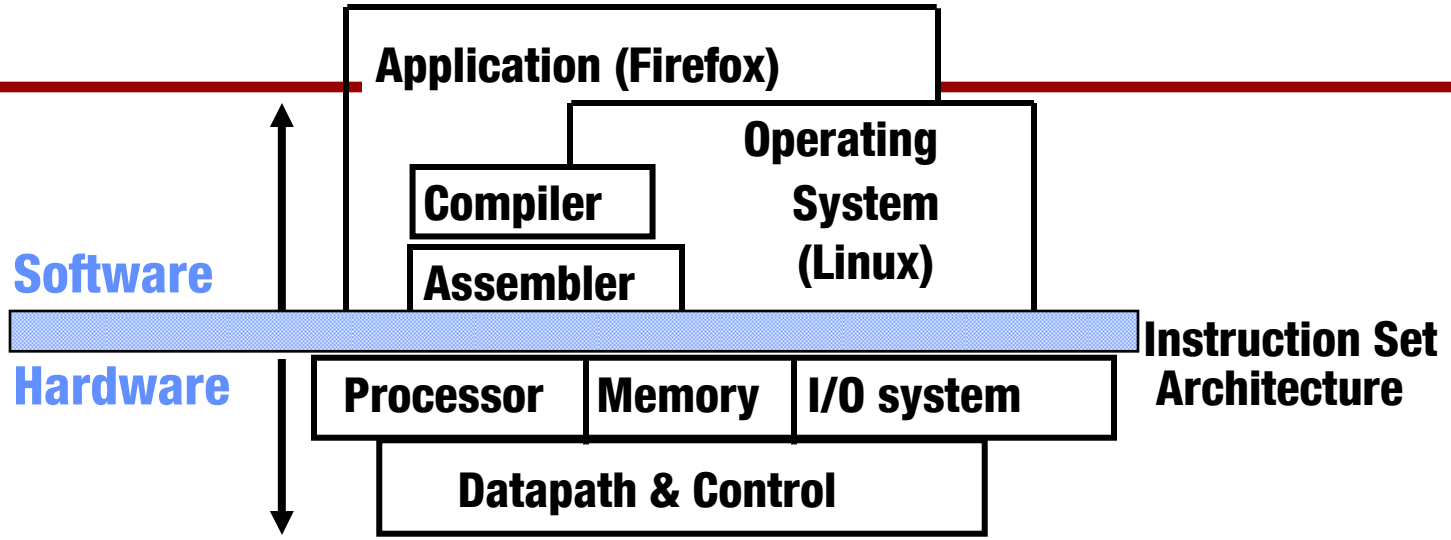
5 components of any Computer



Computers (All Digital Systems) Are At Their Core Pretty Simple

- Computers only work with binary signals
  - Signal on a wire is either 0, or 1
    - Usually called a “bit”
  - More complex stuff (numbers, characters, strings, pictures)
    - Must be built from multiple bits
- Built out of simple logic gates that perform boolean logic
  - AND, OR, NOT, …
- and memory cells that preserve bits over time
  - Flip-flops, registers, SRAM cells, DRAM cells, …
- To get hardware to do anything, need to break it down to bits
  - Strings of bits that tell the hardware what to do are called **instructions**
  - A sequence of instructions called **machine language** program (machine code)

Hardware/Software Interface



- The Instruction Set Architecture (ISA) defines what instructions do
- MIPS, Intel IA32 (x86), Sun SPARC, PowerPC, IBM 390, Intel IA64
  - These are all ISAs
- Many different implementations can implement same ISA (family)
  - 8086, 386, 486, Pentium, Pentium II, Pentium4 implement IA32
  - Of course they continue to extend it, while maintaining binary compatibility
- ISAs last a long time
  - x86 has been in use since the 70s
  - IBM 390 started as IBM 360 in 60s

MIPS ISA

- MIPS – semiconductor company that built one of the first commercial RISC architectures
  - Founded by J. Hennessy
- We will study the MIPS architecture in some detail in this class
- Why MIPS instead of Intel 80x86?
  - MIPS is simple, elegant and easy to understand
  - x86 is ugly and complicated to explain
  - x86 is dominant on desktop
  - MIPS is prevalent in embedded applications as processor core of system on chip (SOC)

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

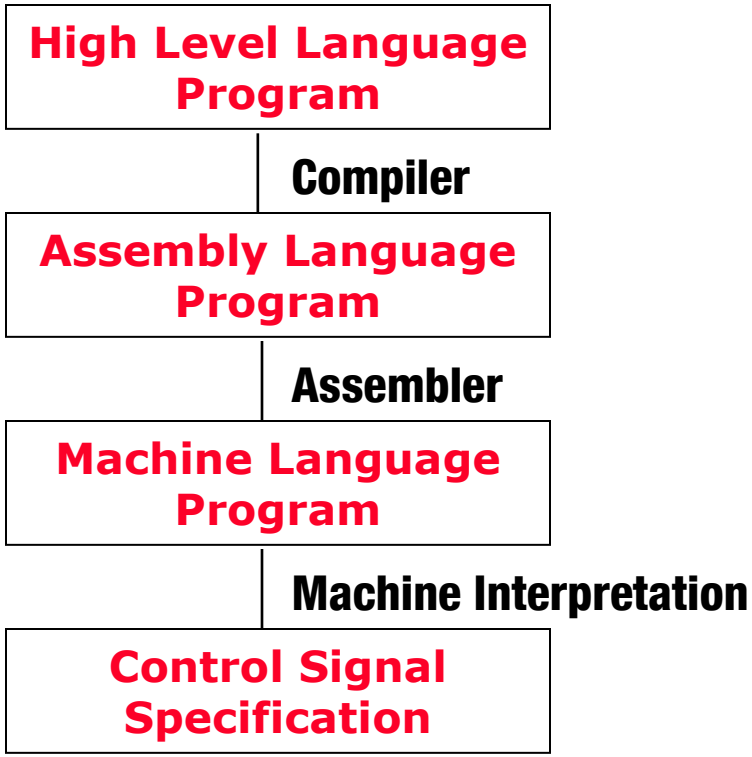
MIPS Processor History

Year	Model - Clock Rate (MHz)	Instruction Set	Cache (I+D)	Transistor Count
1987	R2000-16	MIPS I	External: 4K+4K to 32K+32K	115 thousand
1990	R3000-33		External: 4K+4K to 64K+64K	120 thousand
1991	R4000-100	MIPS III	8K+8K	1.35 million
1993	R4400-150		16K+16K	2.3 million
	R4600-100		16K+16K	1.9 million
1995	Vr4300-133		16K+8K	1.7 million
1996	R5000-200	MIPS IV	32K + 32K	3.9 million
	R10000-200		32K + 32K	6.8 million
1999	R12000-300		32K + 32K	7.2 million
2002	R14000-600		32K + 32K	7.2 million

C vs MIPS I ISA Programmers Interface

- C
  - Variables
    - locals
    - globals
  - Data types
    - int, short, char, unsigned
    - float, double
    - Aggregate data types
    - pointers
  - Operators
    - +, -, *, %,++, <, etc.
  - Control structures
    - If-else, while, do-while, for, switch, procedure call, return
- MIPS I ISA
  - Registers (processor state)
    - 32 32b integer, R0 = 0
    - 32 32b single FP, 16 64b double FP
    - PC and other special registers
  - Memory
    - 2^{32} linear array of bytes
  - Data types
    - word(32b), byte(8b), half-word(16b),
    - single FP(32b), double FP(64b)
  - Operators
    - add, sub, lw, sw, slt, etc.
  - Control
    - Branches, jumps, jump and link

Running An Application

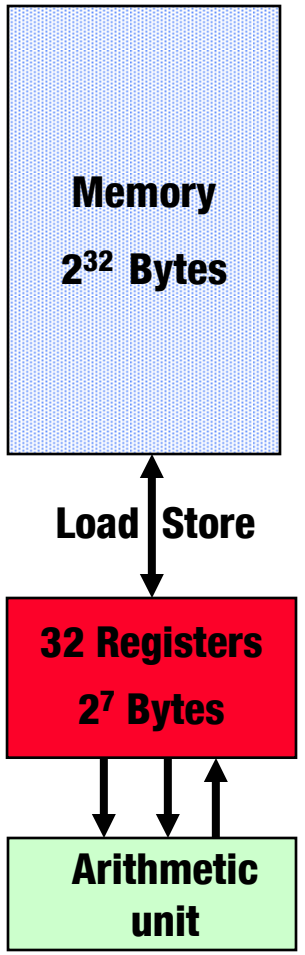



```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
lw  $15,   0($2)
lw  $16,   4($2)
sw $16,   0($2)
sw $15,   4($2)
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

High/Low on control lines

Why Have Registers?

- Memory-memory ISA
  - All HLL variables declared in memory
  - Why not operate directly on memory operands?
  - E.g. Digital Equipment Corp (DEC) VAX ISA
- Benefits of registers
  - Smaller is faster
  - Multiple concurrent accesses
  - Shorter names
- Load-Store ISA
  - Arithmetic operations only use register operands
  - Data is loaded into registers, operated on, and stored back to memory
  - All RISC instruction sets



Using Registers

- Registers are a finite resource that needs to be managed
  - Programmer
  - Compiler: register allocation
- Goals
  - Keep data in registers as much as possible
  - Always use data still in registers if possible
- Issues
  - Finite number of registers available
    - Spill registers to memory when all registers in use
  - Arrays
    - Data is too large to store in registers
- What's the impact of fewer or more registers?

Register Naming

- Registers are identified by a `$<num>`
- By convention, we also give them names
  - **`$zero`** contains the hardwired value 0
  - **`$v0, $v1`** are for results and expression evaluation
  - **`$a0-$a3`** are for arguments
  - **`$s0, $s1, … $s7`** are for save values
  - **`$t0, $t1, … $t9`** are for temporary values
  - The others will be introduced as appropriate
    - See Fig 3.13 p. 140 for details
- Compilers use these conventions to simplify linking

Assembly Instructions

- The basic type of instruction has four components:
  1. Operation name
  2. 1st source operand
  3. 2nd source operand
  4. Destination operand

  - **add dst, src1, src2** **# dst = src1 + src2**

- dst, src1, and src2 are register names ($)
- What do these instructions do?
  - add $1, $1, $1
  - add $1, $1, $1

C Example

```
1: int sum_pow2(int b, int c)
2: {
3:     int pow2 [8] = {1, 2, 4, 8, 16, 32, 64, 128};
4:     int  a, ret;
5:     a = b + c;
6:     if (a < 8)
7:         ret = pow2[a];
8:     else
9:         ret = 0;
10:    return(ret);
11:}
```

Arithmetic Operators

- Consider line 5, C operation for addition

```
a = b + c;
```

- Assume the variables are in registers `$1`-`$3` respectively
- The `add` operator using registers

```
add $1, $2, $3              # a = b + c
```

- Use the `sub` operator for `a=b-c` in MIPS

```
sub $1, $2, $3              # a = b - c
```

- But we know that a,b, and c really refer to memory locations

Complex Operations

- What about more complex statements?

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

- Break into multiple instructions

```
add $t0, $s1, $s2          # $t0 = b + c
add $t1, $t0, $s3          # $t1 = $t0 + d
sub $s0, $t1, $s4          # a = $t1 - e
```

Signed & Unsigned Number

- If given b[n-1:0] in a register or in memory
- Unsigned value

$$value = \sum_{i=0}^{n-1} b_i 2^i$$

- Signed value (2's complement)

$$value = -(b_{n-1} 2^{n-1}) + \sum_{i=0}^{n-2} b_i 2^i$$

Unsigned & Signed Numbers

x	unsigned	signed
0000		
0001		
0010		
0011		
0100		
0101		
0110		
0111		
1000		
1001		
1010		
1011		
1100		
1101		
1110		
1111		

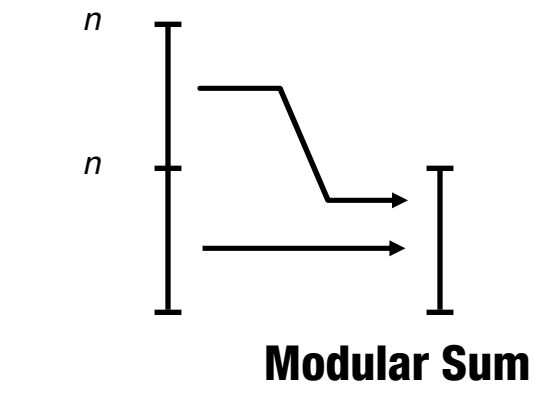
- Example values
  - 4 bits
  - Unsigned: [0, $2^4 - 1$]
  - Signed: [-2^3, $2^3 - 1$]
- Equivalence
  - Same encodings for non-negative values
- Uniqueness
  - Every bit pattern represents unique integer value
  - Not true with sign magnitude

Arithmetic Overflow

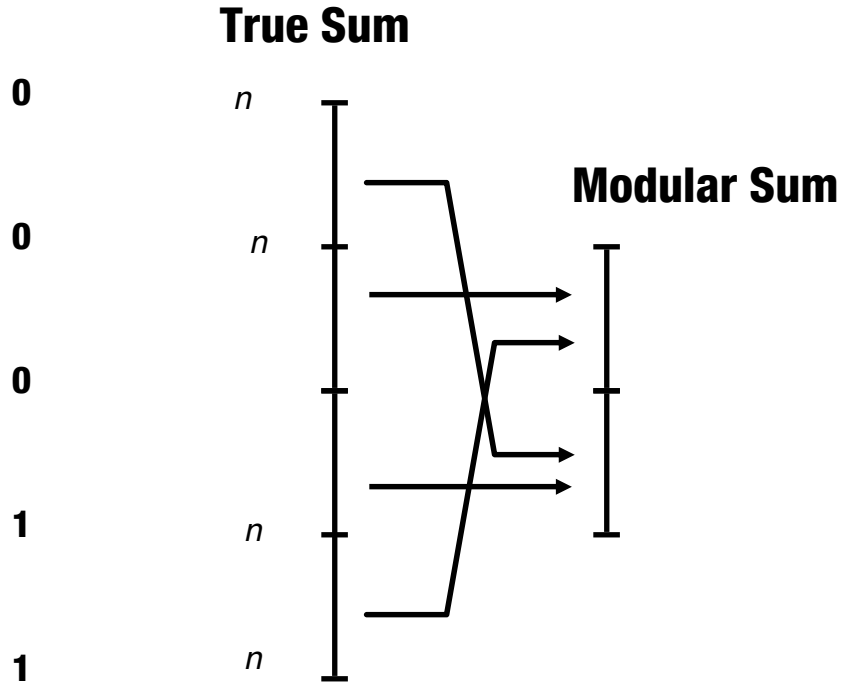
When the sum of two n-bit numbers can not be represented in n bits

Unsigned

- Wraps Around
  - If true sum ≥ 2^n
  - At most once



Signed



Constants

- Often want to be able to specify operand in the instruction: immediate or literal
- Use the `addi` instruction

```
addi dst, src1, immediate
```

- The immediate is a 16 bit signed value between -2^{15} and $2^{15}-1$
- Sign-extended to 32 bits

- Consider the following C code

```
a++;
```

- The `addi` operator

```
addi $s0, $s0, 1            # a = a + 1
```

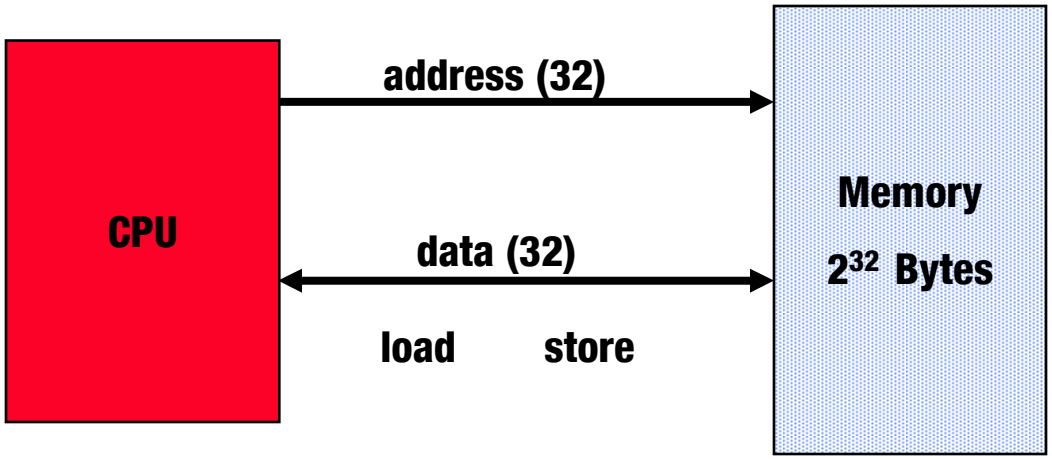
MIPS Simple Arithmetic

Instruction	Example	Meaning	Comments
add	add $1,$2,$3	$1 = $2 + $3	3 operands; Overflow
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; Overflow
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; Overflow
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; No overflow
subtract unsign	subu $1,$2,$3	$1 = $2 – $3	3 operands; No overflow
add imm unsign	addiu $1,$2,100	$1 = $2 + 100	+ constant; No overflow

How does C treat overflow?

Memory Data Transfer

- Data transfer instructions are used to move data to and from memory
- A load operation moves data from a memory location to a register and a store operation moves data from a register to a memory location



Data Transfer Instructions: Loads

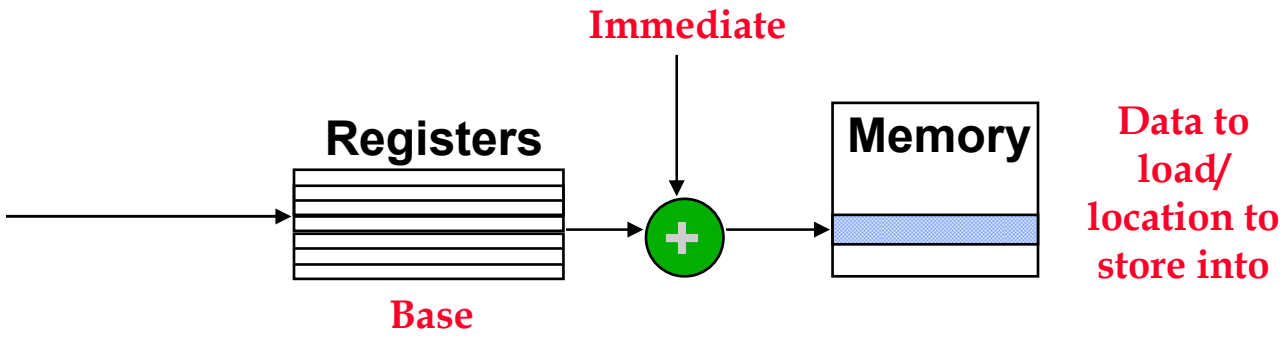
- Data transfer instructions have three parts
  - Operator name (transfer size)
  - Destination register
  - Base register address and constant offset

```
lw dst, offset(base)
```

  - Offset value is a signed constant

Memory Access

- All memory access happens through loads and stores
- Aligned words, half-words, and bytes
- Floating Point loads and stores for accessing FP registers
- **Displacement based addressing mode**



Loading Data Example

- Consider the example

```
a = b + *c;
```

- Use the `lw` instruction to load

  Assume a($s0), b($s1), c($s2)

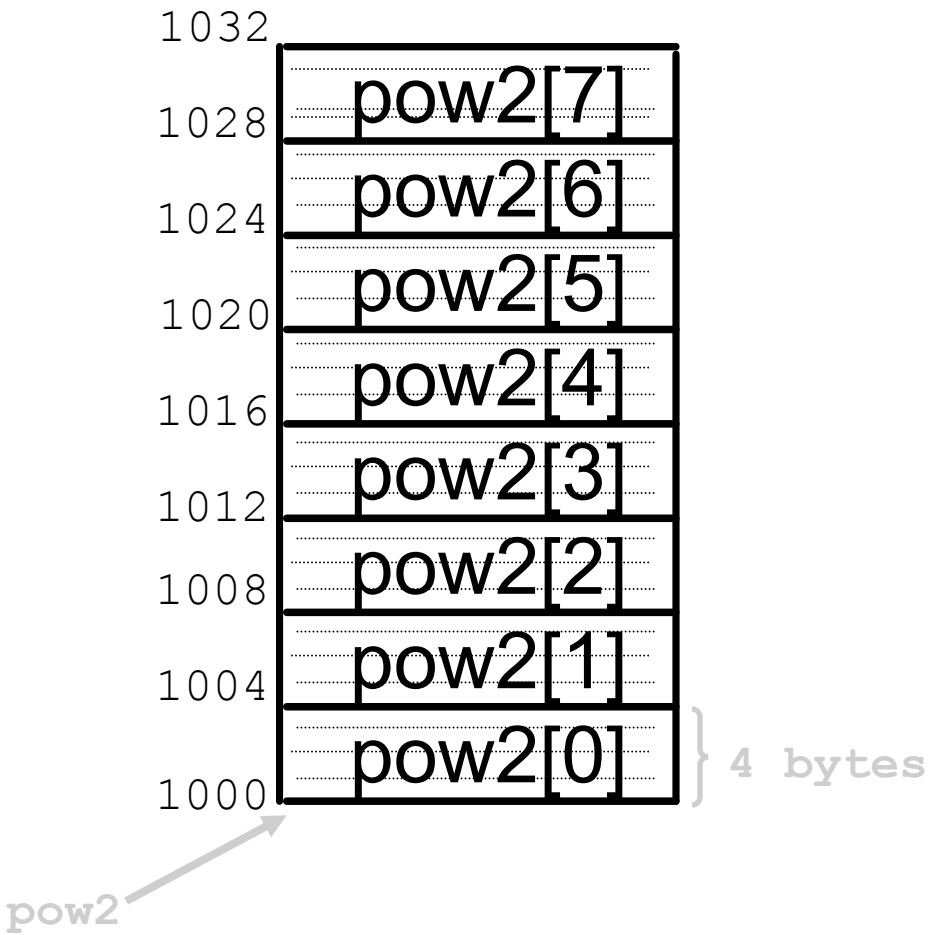
```
lw $t0, 0($s2)          # $t0 = Memory[c]
add $s0, $s1, $t0       # a = b + *c
```

Accessing Arrays

- Arrays are really pointers to the base address in memory
  - Address of element A[0]
- Use offset value to indicate which index
- Remember that addresses are in bytes, so multiply by the size of the element
  - Consider the integer array where `pow2` is the base address
  - With this compiler on this architecture, each int requires 4 bytes
  - The data to be accessed is at index 5: `pow2[5]`
  - Then the address from memory is `pow2 + 5 * 4`

- Unlike C, assembly does not handle pointer arithmetic for you!

Array Memory Diagram



Array Example

- Consider the example

```
a = b + pow2[7];
```

- Use the `lw` instruction offset, assume `$s3 = 1000`

```
lw $t0, 28($s3)      # $t0 = Memory[pow2[7]]
add $s0, $s1, $t0    # a = b + pow2[7]
```

Complex Array Example

- Consider line 7 from sum_pow2()

```
ret = pow2[a];
```

- First find the correct offset, again assume `$s3 = 1000`

```
sll $t0, $s0, 2      # $t0 = 4 * a : shift left by 2
add $t1, $s3, $t0    # $t1 = pow2 + 4*a
lw $v0, 0($t1)       # $v0 = Memory[pow2[a]]
```

Storing Data

- Storing data is just the reverse and the instruction is nearly identical
- Use the **`sw`** instruction to copy a word from the source register to an address in memory

```
sw src, offset(base)
```

- Offset value is signed

Storing Data Example

- Consider the example

```
*a = b + c;
```

- Use the `sw` instruction to store

```
add $t0, $s1, $s2    # $t0 = b + c
sw $t0, 0($s0)       # Memory[s0] = b + c
```

Storing to an Array

- Consider the example

```
a[3] = b + c;
```

- Use the sw instruction offset

```
add $t0, $s1, $s2          # $t0 = b + c
sw $t0, 12($s0)            # Memory[a[3]] = b + c
```

Complex Array Storage

- Consider the example

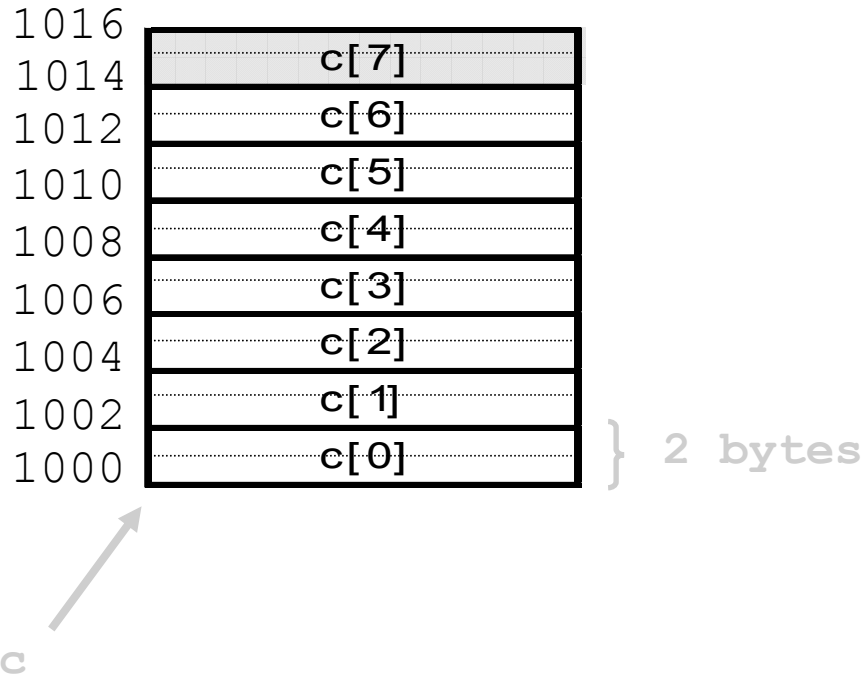
```
a[i] = b + c;
```

- Use the `sw` instruction offset

```
add $t0, $s1, $s2    # $t0 = b + c
sll $t1, $s3, 2      # $t1 = 4 * i
add $t2, $s0, $t1    # $t2 = a + 4*i
sw  $t0, 0($t2)      # Memory[a[i]] = b + c
```

A “short” Array Example

- ANSI C requires a short to be at least 16 bits and no longer than an int, but does not define the exact size
- For our purposes, treat a short as 2 bytes
- So, with a short array c[7] is at c + 7 * 2, shift left by 1

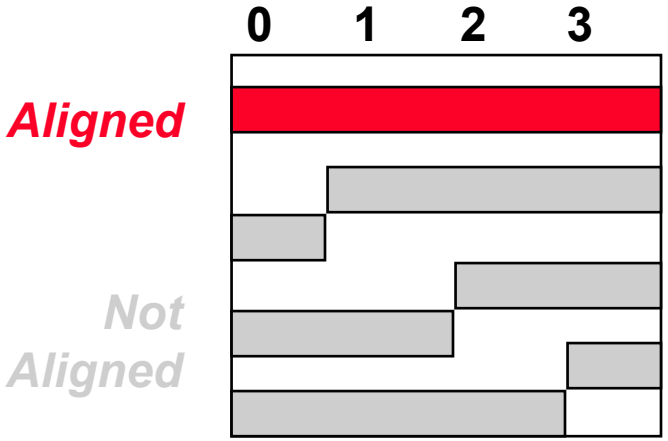


MIPS Integer Load/Store

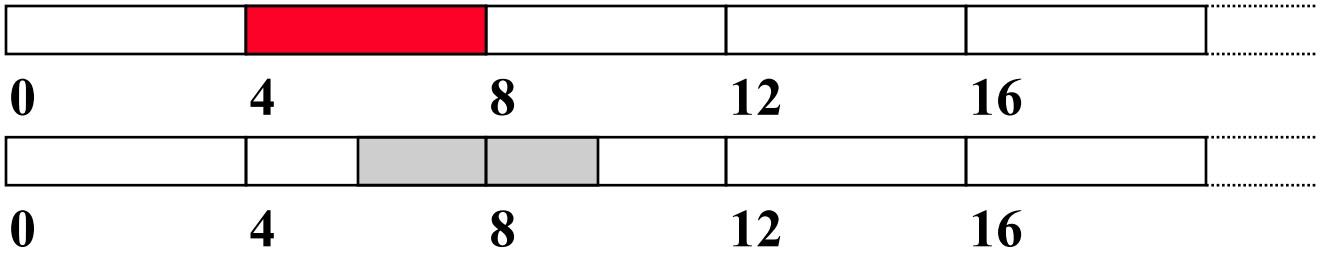
Instruction	Example	Meaning	Comments
store word	sw $1, 8($2)	Mem[8+$2]=$1	Store word
store half	sh $1, 6($2)	Mem[6+$2]=$1	Stores only lower 16 bits
store byte	sb $1, 5($2)	Mem[5+$2]=$1	Stores only lowest byte
store float	sf $f1, 4($2)	Mem[4+$2]=$f1	Store FP word
load word	lw $1, 8($2)	$1=Mem[8+$2]	Load word
load halfword	lh $1, 6($2)	$1=Mem[6+$2]	Load half; sign extend
load half unsign	lhu $1, 6($2)	$1=Mem[8+$2]	Load half; zero extend
load byte	lb $1, 5($2)	$1=Mem[5+$2]	Load byte; sign extend
load byte unsign	lbu $1, 5($2)	$1=Mem[5+$2]	Load byte; zero extend

Alignment Restrictions

- In MIPS, data is required to fall on addresses that are multiples of the data size



- Consider word (4 byte) memory access

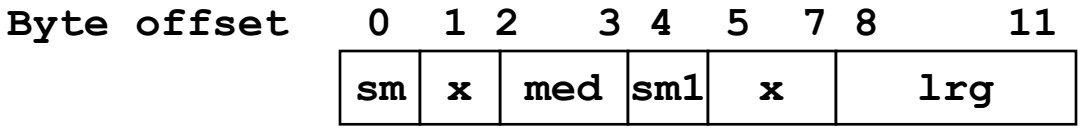


Alignment Restrictions (cont)

- C Example

What is the size of this structure?

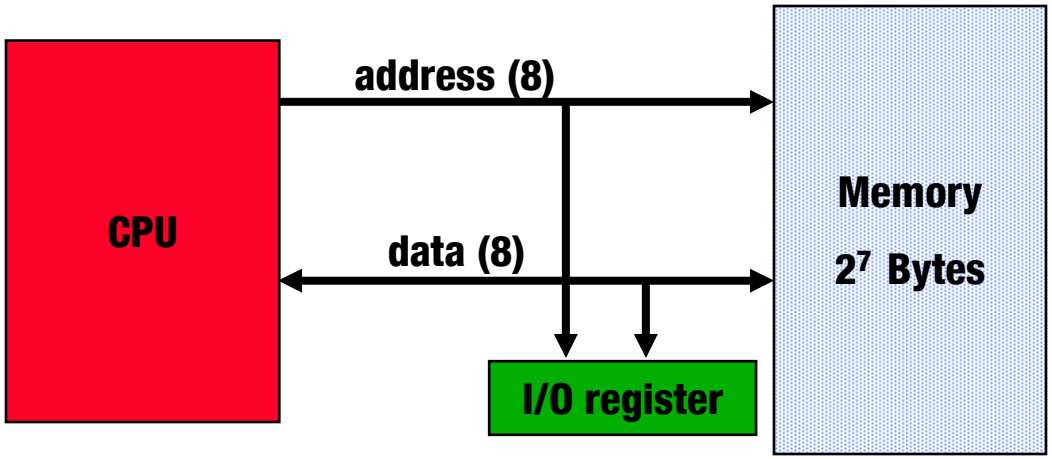
```
struct foo {
        char sm;
        short med;
        char sm1;
        int lrg;
}
```



- Historically
  - Early machines (IBM 360 in 1964) required alignment
  - Removed in 1970s to reduce impact on programmers
  - Reintroduced by RISC to improve performance
- Also introduces challenges with memory organization with virtual memory, etc.

Memory Mapped I/O

- Data transfer instructions can be used to move data to and from I/O device registers
- A load operation moves data from a an I/O device register to a CPU register and a store operation moves data from a CPU register to a I/O device register



I/O register at address 0x80 (128)

Changing Control Flow

- One of the distinguishing characteristics of computers is the ability to evaluate conditions and change control flow
- C
  - If-then-else
  - Loops
  - Case statements
- MIPS
  - Conditional branch instructions are known as *branches*
  - Unconditional changes in the control flow are called *jumps*
  - The target of the branch/jump is a label

Conditional: Equality

- The simplest conditional test is the `beq` instruction for equality

```
beq reg1, reg2, label
```

- Consider the code

```
        if (a == b) goto L1;
        // Do something
L1:     // Continue
```

- Use the `beq` instruction

```
        beq $s0, $s1, L1
        # Do something
L1:     # Continue
```

Conditional: Not Equal

- The `bne` instruction for not equal

```
bne reg1, reg2, label
```

- Consider the code

```
        if (a != b) goto L1;
        // Do something
L1:     // Continue
```

- Use the `bne` instruction

```
        bne $s0, $s1, L1
        # Do something
L1:     # Continue
```

Unconditional: Jumps

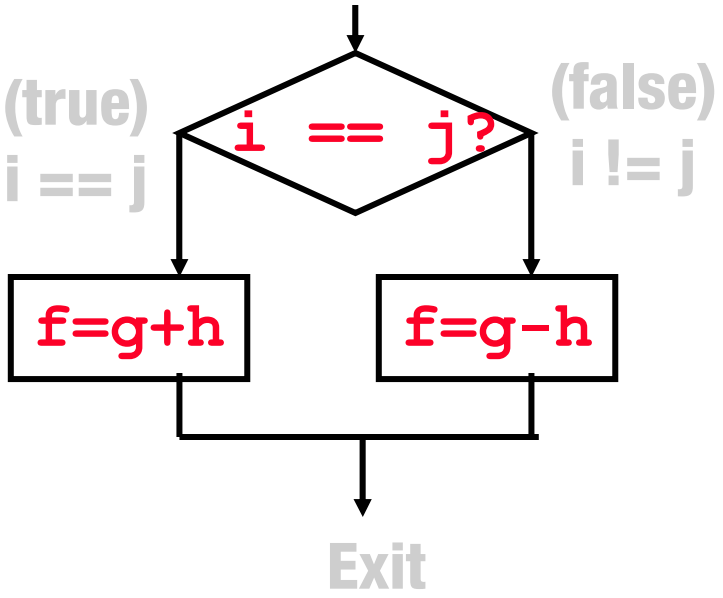
- The `j` instruction jumps to a label

```
j label
```

If-then-else Example

- Consider the code

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



If-then-else Solution

- Create labels and use equality instruction

```
        beq $s3, $s4, True   # Branch if i == j
        sub $s0, $s1, $s2    # f = g - h
        j Exit               # Go to Exit
True:   add $s0, $s1, $s2    # f = g + h
Exit:
```

Other Comparisons

- Other conditional arithmetic operators are useful in evaluating conditional expressions using <, >, <=, >=
- Register is “set” to 1 when condition is met

- Consider the following C code

```
if (f < g) goto Less;
```

- Solution

```
slt $t0, $s0, $s1          # $t0 = 1 if $s0 < $s1
bne $t0, $zero, Less       # Goto Less if $t0 != 0
```

MIPS Comparisons

Instruction	Example	Meaning	Comments
set less than	`slt $1, $2, $3`	$1 = ($2 < $3)	comp less than signed
set less than imm	`slti $1, $2, 100`	$1 = ($2 < 100)	comp w/const signed
set less than uns	`sltu $1, $2, $3`	$1 = ($2 < $3)	comp < unsigned
set l.t. imm. uns	`sltiu $1, $2, 100`	$1 = ($2 < 100)	comp < const unsigned

- C

```
if (a < 8)
```

- Assembly

```
slti $v0,$a0,8              # $v0 = a < 8
beq  $v0,$zero, Exceed      # goto Exceed if $v0 == 0
```

C Example

```
1: int sum_pow2(int b, int c)
2: {
3:     int pow2 [8] = {1, 2, 4, 8, 16, 32, 64, 128};
4:     int  a, ret;
5:     a = b + c;
6:     if (a < 8)
7:         ret = pow2[a];
8:     else
9:         ret = 0;
10:    return(ret);
11:}
```

```
sum_pow2:                                   # $a0 = b, $a1 = c
          addu   $a0,$a0,$a1                # a = b + c, $a0 = a
          slti   $v0,$a0,8                  # $v0 = a < 8
          beq    $v0,$zero, Exceed          # goto Exceed if $v0 == 0
          addiu  $v1,$sp,8                  # $v1 = pow2 address
          sll    $v0,$a0,2                  # $v0 = a*4
          addu   $v0,$v0,$v1                # $v0 = pow2 + a*4
          lw     $v0,0($v0)                 # $v0 = pow2[a]
          j      Return                     # goto Return
Exceed:   addu   $v0,$zero,$zero            # $v0 = 0
Return:   jr     ra                         # return sum_pow2
```