Computer Architecture

Chapter 2
Instructions: Language of the Computer

Fall 2005

Department of Computer Science

Kent State University

Assembly Language

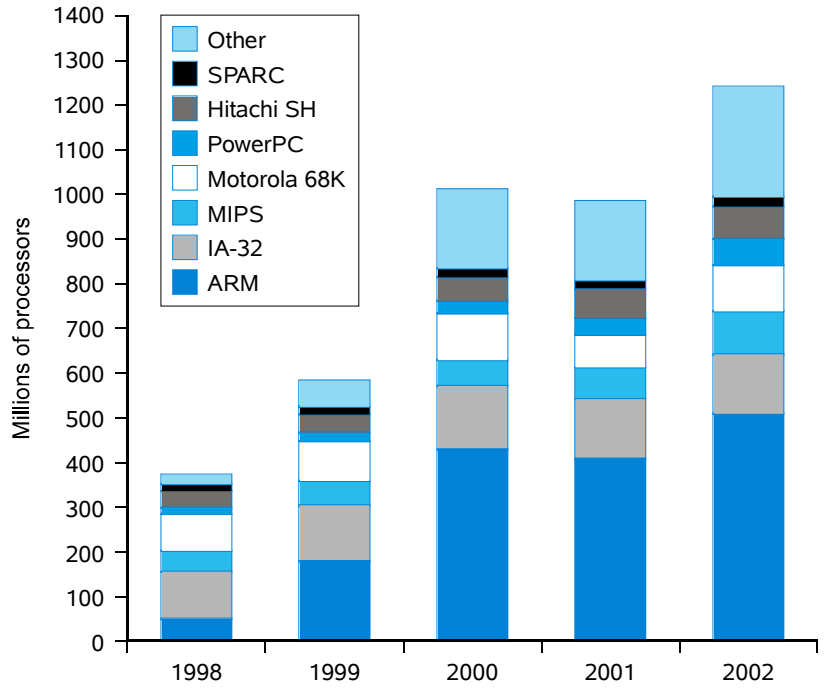
- Encodes machine instructions using symbols and numbers instead of 0's and 1's
- Every processor architecture has its own assembly language
  - Syntax is usually similar
  - Instruction set is main difference
- Assembly language code is not portable; should be used only when needed

Reasons for Learning AL

- Operating systems and device drivers
- Compilers
- Time-critical code
- Using special processor features
- Understanding what's going on behind the scenes

MIPS

- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - Almost 100 million MIPS processors manufactured in 2002
  - used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, …



MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

Example:

C code: `a = b + c`

MIPS 'code': `add a, b, c`

"The natural number of operands for an operation like addition is three… requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple"

More Operands

- If we need more operands then we have to use multiple instructions
- Example: a = b + c + d;
  - add a, b, c
  - add a, a, d
- Note that an operand can be both a source and a destination (just like C)

Complex Operations

- Consider: f = (g + h) - (i + j)
- Must first evaluate expressions inside parentheses then subtract
- Use temporary variables to store intermediate results in complex expressions
- In assembly language:
  - add t0, g, h
  - add t1, i, j
  - sub f, t0, t1
- *t0* and *t1* are temporaries

Register

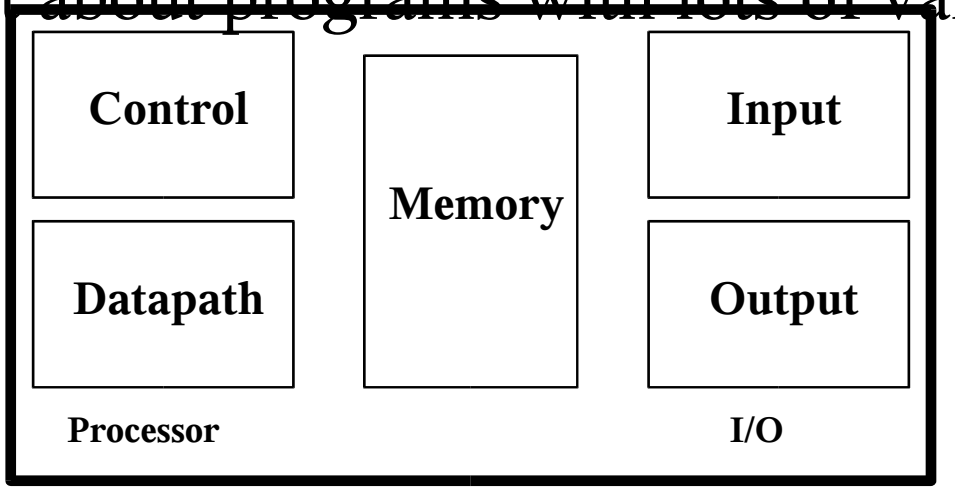
- Instruction operands must be stored in *registers*
- Registers are like memory cells, but they are much faster and there are very few of them
- MIPS has 31 general-purpose registers
- Each register is 32 bits wide
- In assembly language, a register is designated with a dollar sign ($)
- For now we will use registers $s0...$s7 for holding variables and registers $t0...$t9 as temporaries

Register Operands

- Let's consider our complex expression once again: f = (g + h) - (i + j)
- First we must decide which register to assign to each variable
  - *f* is $s0, *g* is $s1, *h* is $s2, *i* is $s3, and *j* is $s4
- Replace variable names in the previous code with the corresponding register name
  - add $t0, $s1, $s2
  - add $t1, $s3, $s4
  - sub $s0, $t0, $t1

Registers vs. Memory

- Arithmetic instructions operands must be registers,
  — only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables

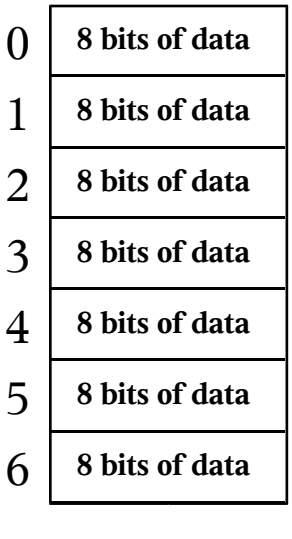


Memory

- There aren't enough registers to hold all the variables used by most programs
- Complex variables like structures and arrays cannot be stored in registers either
- In these cases we must use memory
- However, arithmetic instructions require their operands to be in registers
- *Data transfer instructions* move data between memory and registers

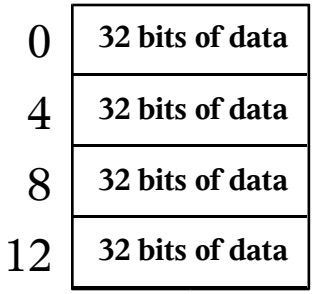
Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.



Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.



Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$
- Words are aligned

  i.e., what are the least 2 significant bits of a word address?

Two Bonus Questions

- Why we use byte=8 bits?
- Endian question:
  - Big Endian
  - Little Endian
  - Which is better?
  - What does it mean to us?
  - http://www.cs.umass.edu/~verts/cs32/endian.html

Load

- The **lw** (load word) instruction transfers a full 32-bit word from memory to a register
- Syntax: **lw** *dest*, *offset***(***base***)**
- The offset (a constant) is added to the value of the base register to generate the address
- The data word located at that address is placed in the destination register
- How many operands?

Accessing an Array Element

- To access an element of an array you need three things
  - Base address
  - Index
  - Width of the base type
- Formula: address + (index × width)
- How can we use the **lw** instruction for this?
  - Place the base address in a register
  - Compute index × width and use that as the offset
  - The unit of width is byte, why?
  - Byte-Addressing

Store

- The **sw** (store word) instruction stores a full 32-bit word from a register to memory
- It works just like **lw** except that the destination register is replaced with a source register
- Syntax: **sw** *src*, *offset***(***base***)**
- The value in the source register is stored in memory at the computed address

Load and store instructions

- Example:

  C code: `A[12] = h + A[8];`

  Assume A is integer array (4 bytes) and base address is store in $s3 and variable h is stored in $s2.

  MIPS code:

```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 48($s3)
```

- Store word has destination last
- Remember **arithmetic operands** are registers, not memory!

  Can't write: `add 48($s3), $s2, 32($s3)`

So far we've learned:

- MIPS
  - — loading words but addressing bytes
  - — arithmetic on registers only

- Instruction — Meaning

```
add $s1, $s2, $s3          $s1 = $s2 + $s3
sub $s1, $s2, $s3          $s1 = $s2 - $s3
lw $s1, 100($s2)   $s1 = Memory[$s2+100]
sw $s1, 100($s2)   Memory[$s2+100] = $s1
```

Immediate Operands

- Many times the operands to an arithmetic operation are small constants
  - a=a+9
- It is cumbersome and inefficient to have to load these constants from memory
- Some arithmetic operations permit an *immediate* (constant) operand to be encoded directly in the instruction
- Example: **addi** (add immediate)
- The immediate operand must be the second source

Zero Register

- The constant value zero is used extensively throughout most programs
- MIPS has a special register (**$zero** or **$0**) which is hardwired to zero
- Attempts to change the value of **$zero** are ignored
- Commonly used to synthesize instructions that MIPS does not support directly
  - Negate: **sub** *dest*, **$zero**, *src*
- This is why we say MIPS has 31 general-purpose registers instead of 32

A Research Question

- Suppose a program will use a large number of memory, and we only have a very small number of registers
- Suppose at certain stage, the program has used all the registers and a new variable need to load into the register, what should we do?
- Think about the case where a register might be used by later stage of the program?

Review (1)

- In MIPS, how many operands per instruction?
- How many registers in MIPS?
- How many bits per register?
-
```
add $s1, $s2, $s3
sub $s1, $s2, $s3
lw $s1, 100($s2)
sw $s1, 100($s2)
addi $s1, $s2, 8
```

Review (2)

- C code: `A[16] = h - A[4];`

  Assume A is integer array (4 bytes) and base address is store in $s1 and variable h is stored in $s2.

  MIPS code:

```
lw  $t0, 16($s1)
sub $t0, $s2, $t0
sw  $t0, 64($s1)
```

Bit Operations

- Arithmetic operations treat the contents of a register as a binary number
- Bit operations work on the individual bits without interpreting them in any way
- We can use bit operations when the content of a register is not actually a binary number (bit fields)
- By exploiting the properties of binary numbers we can perform some arithmetic operations faster

Shift Operations (1)

- Shift instructions move the bits in a word to the left or to the right
- There are two shift instructions:
  - Shift left logical (**sll**)
  - Shift right logical (**srl**)
- Syntax: **sll** *dest*, *src*, *shamt*
- *shamt* is the shift amount (number of bits) and must be between 0 and 31

Shift Operation (2)

- sll $t1,$s0,4
  srl $t2,$s0,2

  $s0=9

  $s0=31

  What's in $t1, and $t2?

Logical Operations (1)

- Logical operations combine the bits from two words using a Boolean operator (AND, OR, etc.)
- **and** and **andi** can isolate a subset of bits in a word
- **or** and **ori** can combine parts of words together
- There is no NOT instruction in MIPS since that would violate the two source, one destination format, instead we have **nor**

Logic Operations (2)

- $s2=(D00)_{16}$, $s3=(3C00)_{16}$
- and $s1,$s2,$s3

  or $s1,$s2,$s3

  nor $s1,$s2,$s3

  andi $s1,$s2,100

  ori $s1,$s2,100

- You can synthesize NOT using **nor, how?**

Bit Manipulation Tricks

- Multiply by 2^n
  - Left shift of *n* bits
- divide by 2^n
  - Right shift
- Copy contents one register to another
  - **or** *dest*, **$zero**, *src*
- Copy an immediate value into a register
  - **ori** *dest*, **$zero**, *immed*

Branches and Jumps

- Normally the processor executes instructions sequentially
- *Branch* and *jump* instructions allow us to skip over instructions or to go back and repeat them
- Branches can be *conditional* or *unconditional*
- A conditional branch whose condition is true is said to be *taken* otherwise it is *untaken* (*not taken*)
- An untaken branch *falls through* to the next sequential instruction

Branch Instructions

- Branch if equal (**beq**) compares two registers and branches if they are equal
  - Syntax: **beq** *src1*, *src2*, *label*
- Branch if not equal (**bne**) is similar but branches when the registers are not equal
  - Syntax: **bne** *src1*, *src2*, *label*
- Jump (**j**) is an unconditional branch
  - Syntax: **j** *label*

Labels

- Branch instructions specify where to go using a *label* as an operand
- The label must be defined somewhere in the program by attaching it to the front of another instruction
- Example:
  - label**: add** $s0, $s1, $s2

If Statement

- if (expr)
  { }
- Use a conditional branch to skip past the *then* block when the condition is false
- When the condition is true the branch won't be taken (it will fall through) and the *then* block will be executed
- Note that the branch should be taken when the condition is false so the branch instruction will be the opposite of the condition

If Example

- In C

```
if (a == b) {
    z = 0;
}
z++;
a -> $s0, b-> $s1, z->$s2
```

- In MIPS assembly language

```
bne $s0, $s1, endif
or $s2, $zero, $zero
endif: addi $s2, $s2, 1
```

If-Else Statement

- if (expr)
  { }
  else
  { }
- Use a conditional branch to skip to the *else* block when the condition is false
- When the condition is true the branch won't be taken and the *then* block will be executed
- Use an unconditional branch at the end of the *then* block to skip over the *else* block

If-Else Example (1)

- Example:

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

(f, g, h, i, j → registers: $s0, $s1, $s2, $s3, $s4)

```
bne $s3, $s4, Else          #go to Else if i !=  j
add $s0, $s1, $s2
j Exit
Else: sub  $s0, $s1, $s2
Exit:
```

If-Else Example (2)

- In C++

```
if (a == b) {
    z = 1;
}
else {
    z = A[1];
}
z++;
```

a -> $s0, b-> $s1,
z->$s2, A->$s7

- In assembly language

bne $s0, $s1, else
ori $s2, $zero, 1
j endif
else: **lw** $s2, 4($s7)
endif: **addi** $s2, $s2, 1

Review (1)

- Arithmetic operation
  - add, sub, addi
- Memory operation
  - lw, sw
- Bit shifting operation
  - sll, srl
- Logic operation
  - and, or, nor, andi, ori
- Branch operation
  - beq, bne, j

How to implement NOT operation? ($s3=NOT($s3))

Review (2)

- How to copy one register ($s1) to another register ($s0) or copy a constant (200) into one register ($s0)?
- How to multiply a register ($s2) by 16?
-
```
if (a != b) {
    z = A[8]*4;
}
else {
    z = 0;
}
z--;
```

a -> $s0, b-> $s1,
z->$s2, A->$s7
A is integer array.

Loops

- While
  - while (expr)
    { }
  - Evaluate condition
  - If condition is false go to the end
  - Execute loop body
  - Go back to the beginning
- Do-While
  - do
    { }
    while (expr);
  - Execute loop body
  - Evaluate condition
  - If condition is true go back to the beginning

While Example

- In C++

```
while (a = = b) {
    z++;
}
z = a;
```

a -> $s0, b-> $s1,
z->$s2

•In assembly language

```
loop:
    bne $s0, $s1, endwhile

    addi $s2, $s2, 1
    j loop
endwhile:
    or $s2, $s0, $zero
```

Do-While Example

- In C

```
do {
    z--;
} while (a == b);
z = b;
```

a -> $s0, b-> $s1, z->$s2

- In assembly language

```
loop: addi $s2, $s2, -1
beq $s0, $s1, loop
or $s2, $s1, $zero
```

Comparisons (1)

- Set on less than (**slt**) compares its source registers and sets its destination register to 1 if *src1* < *src2* and to 0 otherwise
  - Syntax: **slt** *dest*, *src1*, *src2*
- Set on less than immediate (**slti**) perform the same comparison, but its second source operation is an immediate value
  - Syntax: **slti** *dest*, *src1*, *immed*
- Combined with **beq** and **bne** these instructions can implement all possible relational operators

Comparisons (2)

```
 if  $s3 < $s4 then
      $t0 = 1
else
     $t0 = 0


if  $s3 < 0 then
      $t0 = 1
else
     $t0 = 0
```

Relational Branches (1)

- Branch if less than
  - if (src1 < src2) goto label;
  - **slt** $t0, *src1*, *src2*
  - **bne** $zero, $t0, *label*
- Branch if greater than or equal to
  - if (src1 >= src2) goto label;
  - **slt** $t0, *src1*, *src2*
  - **beq** $zero, $t0, *label*

Relational Branches (2)

- Branch if greater than
  - if (src1 > src2) goto lable;
  - **slt** $t0, *src2*, *src1*
  - **bne** $zero, $t0, *label*
- Branch if less than or equal
  - if (src1 <= src2) goto label;
  - **slt** $t0, *src2*, *src1*
  - **beq** $zero, $t0, *label*

So far, we have learned

- How to express
  - If-Then
  - If-Then-Else
  - While
  - Do-While
- How to express
  - >, <, >=, <=

By using Branch/Jump operation
  - beq, bne,
  - j

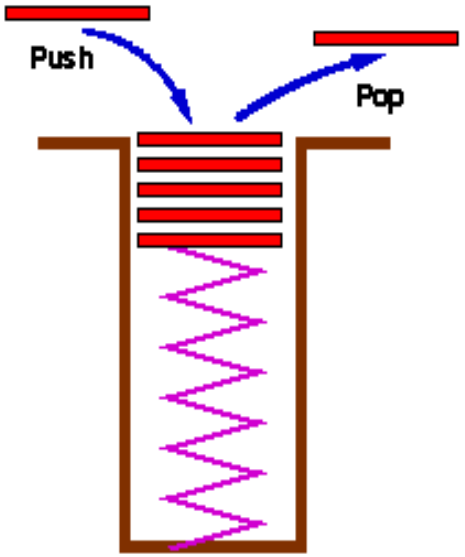
and Comparison operation
  - slt

Case/Switch Statement

- Can be implemented like a chain of if-then-else statements
- Using a *jump address table* is faster
- Must be able to jump to an address loaded from memory
- Jump register (**jr**) gives us that ability
- Syntax: **jr** *src*

Data Structure: Stack

- Last In-First Out
- Push, Pop, top
- Push(5), Push(6), Pop; Push(4);



- Data that a function needs to save in memory is *pushed* onto the stack
- Before returning, the function *pops* the data off of the stack

MIPS support for stack

- In MIPS, the *stack pointer* ($sp) contains the address of the last word pushed onto the stack
- The stack grows downward from higher addresses to lower ones

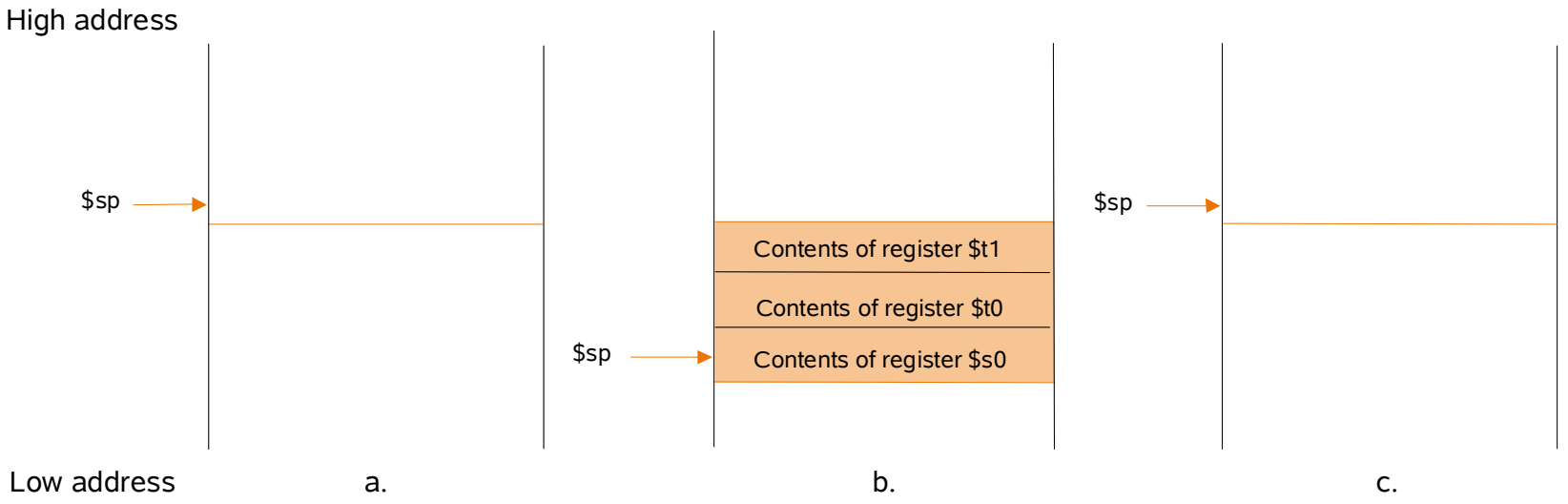
Accessing the Stack

- Push a word onto the stack
  - **addi** $sp, $sp, -4
  - **sw** *src*, 0($sp)
- Pop a word off of the stack
  - **lw** *dest*, 0($sp)
  - **addi** $sp, $sp, 4
- Typically we batch multiple pushes and pops together

Accessing the Stack (cont'd)

- Push three words onto the stack
  - **addi** $sp, $sp, -12
  - **sw** $t0, 8($sp)
  - **sw** $t1, 4($sp)
  - **sw** $s0, 0($sp)
- Pop three words off of the stack
  - **lw** $s0, 0($sp)
  - **lw** $t1, 4($sp)
  - **lw** $t0, 8($sp)
  - **addi** $sp, $sp, 12

Stack Illustration



Functions

- A function (or procedure) is a sequence of instructions that can be used to perform a task
- Functions may accept parameters (arguments) and/or return a value
- A function can also define local variables for its use which exist only until the function returns
- Functions can call other functions
- When a function is *called* control transfers to the function; once the function completes its task it *returns*

Function Example (1)

- In C++:

```
int leaf_example( int g, int h, int i, int j)  {
    int f;
    f= (g+h)-(i+j);
    return f;
 }

 x=leaf_example (g1,h1,i1,j1);
 y=leaf_example (5,2,3,4);
```

Parameters/returning value/call+return/local variables

1) In MIPS: $a0-$a3, four argumented registers to pass parameters
2) $v0-$v1: two value register in which to return values
3) call+return

Call and Return

- Call and return are nothing more than unconditional jumps
- The calling function (the *caller*) jumps to the beginning of the function
- The called function (the *callee*) jumps back to the point from which it was called (*return address*)
- Since a function can be called from any number of places the return address changes with each call

Jump and Link

- The jump and link (**jal**) instruction performs an unconditional jump just like **j**, but first saves the address of the next instruction in the return address register (**$ra**)
  - **jal** *label*
- A function returns by jumping to the address stored in that register:
  - **jr** $ra

Example (1)

- Calling a function:
  - **jal** function1
  - **and** $s0, $s1, $s2
- Function:
  - function1: **add** $t0, $t1, $t2
  - **sub** $t3, $t4, $t5
  - **jr** $ra

Function Example (2)

- In C++:

```
int leaf_example( int g, int h, int i, int j)  {
    int f;
    f= (g+h)-(i+j);
    return f;
}
```

4) How to deal with local variables/ local registers

MIPS code

```
leaf_example:
    addi $sp,$sp, 12
    sw $t1, 8($sp)
    sw $t0, 4($sp)
    sw $s0, 0($sp)

   add $t0,$a0,$a1
   add $t1,$a2,$a3
   sub $s0,$t0,$t1

    add $v0,$s0,$zero

     lw $s0, 0($sp)
     lw $t0, 4($sp)
     lw $t1, 8($sp)
     addi $sp, $sp, 12

     jr $ra
```

```
x=leaf_example(5,4,3,2)

ori $a0, $zero, 5
ori $a1, $zero, 4
ori $a2, $zero, 3
ori $a3, $zero, 2

jal leaf_example

ori $s0, $v0, $zero
```

(x is in s0)

Review (1)

- Arithmetic operation
  - add, sub, addi
- Memory operation
  - lw, sw
- Bit shifting operation
  - sll, srl
- Logic operation
  - and, or, nor, andi, ori
- Branch operation
  - beq, bne, j
- Comparison operation
  - slt, slti
- Jumps for calling/return
  - jal, jr

Review (2)

- What do the following two instructions do?
  - **slt** $t0, *src1*, *src2*
  - **beq** $zero, $t0, *label*
- What are the basic support for stack in MIPS?
  - stack pointer
  - stack direction
- Using MIPS language to push ($s0), push ($t0), push ($t1), and then pop → $t2.

Functions in MIPS (1)

- What are the basic steps to call a function?
  - Assign parameters ($a0, $a1, $a2, $a3)
  - Call the function (jal)
  - Copy the results for the returning value registers

Functions in MIPS (2)

- What are the basic steps in a function in MIPS?
  - Save registers will be used the function (stack/push)
  - Execute the body of a function
  - Copy the results the returning-value registers ($v0, $v1)
  - Recover registers used by this function (stack/pop)
  - Jump back

Example

- In C++:

```
int max ( int g, int h, int i)  {
    int f;
    int f1;
    f1=g+h;
    if ( f1> f2)
        f=f1;
   else
       f=i;
    return f;
  }
```

```
x=max(y,2,3);
```

x,y is in register $s0, $s1
f1, f use register $t0, $s0

Register Use

- *Callee-saved* registers are owned by caller; if a function uses a callee-saved register it must save the old value and restore that value before it returns
- *Caller-saved* registers are owned by the callee; if a caller is using a caller-saved register, it must save the old value before calling the function and restore it after the function returns
- In MIPS, $s0-$s7 are callee-saved and $t0-$t9 are caller-saved

MIPS Registers

Name	Number	Usage	Saved on Call?
$zero	0	Zero	N/A
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Return Value/Temporaries	No
$a0-$a3	4-7	Arguments	No
$t0-$t9	8-15, 24-25	Temporaries	No
$s0-$s7	16-23	Saved	Yes
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

Instruction Formats

- R-type
  - Arithmetic, logic, and shift instructions
  - All register operands
- I-type
  - Arithmetic and logic with immediates
  - Conditional branch instructions
  - Load and store instructions
- J-type
  - Jump and jump-and-link

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- op: operation code or *opcode*
- rs: first source register
- rt: second source register
- rd: destination register
- shamt: shift amount
- funct: function code

Machine Language Instructions

Arithmetic Instructions

- MIPS Instructions:
  - Example: add $t0, $s1, $s2 # $t0 = $s1 + $s2
- Registers represented by numbers:
  - $t0 -> 8; $s1 -> 17; $s2 -> 18
  - Instruction Format (R-Type):

op	rs	rt	rd	shamt	funct
0	17	18	8	0	32

Machine Language Instructions

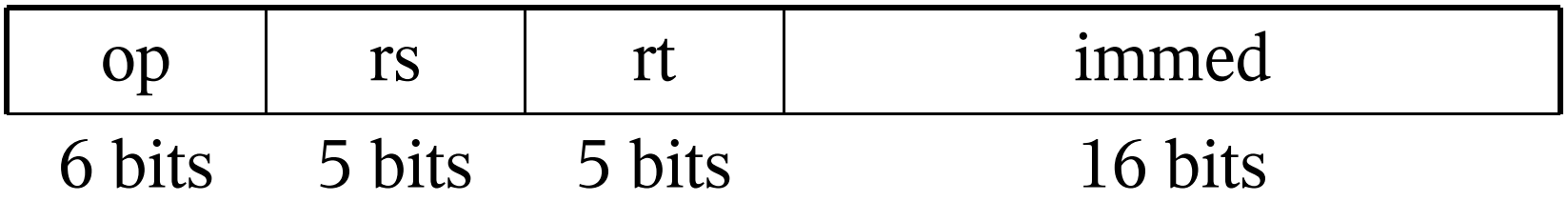
Arithmetic Instructions

add $t0, $s1, $s2

op	rs	rt	rd	shamt	funct
0	17	18	8	0	32
000000	10001	10010	00100	0	100000

MIPS Instructions: 32 bits long

I-Type



- op: opcode
- rs: first source register
- rt: second source register or destination register
- immed: immediate value or branch displacement

Machine Language Instructions

Data Transfer Instructions

- MIPS Instructions:
  - Example: lw $t0, 32($3) # $t0 gets A[8]
- Registers represented by numbers:
  - Instruction Format (I-Type):

op	rs	rt	16 bit number

35	18	9	32

I-Type Instructions: 32 bits long

Example

- Example: C- code: A[300] = h + A[300];

- MIPS code:

```
lw   $t0, 1200 ($t1)
add $t0,  $s2,  $t0
sw   $t0, 1200($t1)
```

op	rs	rt	rd	addr/ shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Machine Language

- Example: C- code: A[300] = h + A[300];
- Machine Language

op	rs	rt	rd	addr/ shamt	funct
10011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	0000	100000
101011	01001	01000	0000 0100 1011 0000		

Machine Language Instructions

Branch Instructions

- Instruction Format (J -Type):

J -Type Instructions: 32 bits long

R-Type Example

- In assembly: **add** $s0, $s1, $s2

0	17	18	16	0	32
op	rs	rt	rd	shamt	funct

000000	10001	10010	10000	00000	100000
op	rs	rt	rd	shamt	funct

I-Type Example

- In assembly: **addi** $s0, $s1, 5

8	17	16	5
op	rs	rt	immed

001000	10001	10000	0000000000000101
op	rs	rt	immed

How Branches Work

- The *program counter* (PC) is a register that contains the address of the instruction that the processor is currently executing
- Most of time the processor increments of the value of PC by 4 after executing an instruction
- Branch instructions allow the PC to be modified in other ways

Computing Branch Targets

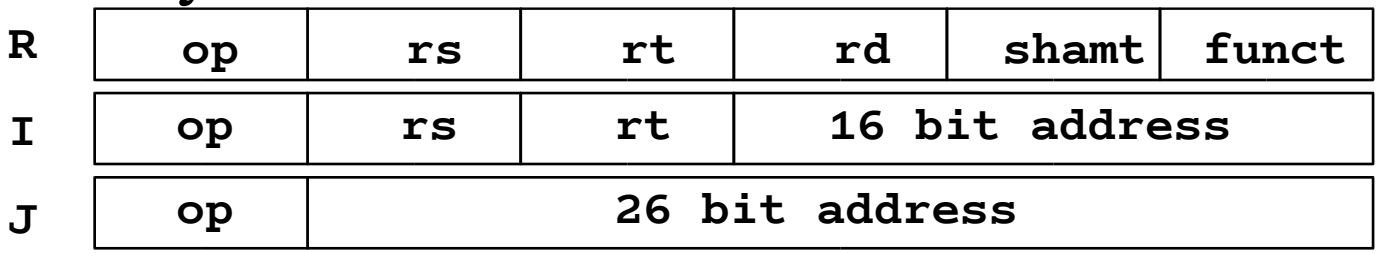
- For J-type instructions (**j** and **jal**) the lower bits of the target address are taken directly from the instruction
  - This is called *pseudodirect addressing*
- For I-type instructions (**beq** and **bne**) the immediate field is added to the PC
  - This is called *PC-relative addressing*
- **jr** simply copies the register's value into the PC

Branching Far Away

- An immediate value is limited to 16 bits
- The target of a conditional branch must be within 32,767 instructions of the branch instruction itself
- To branch farther, use two instructions:
  - **bne** $s0, $s1, L2
  - **j** L1
  - L2: ...

Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats



- rely on compiler to achieve performance
  — what are the compiler's goals?
- help compiler where we can

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 (clock cycles per instruction)

–***"The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions"***

- For example: IA-32