Instructions: Language of the Computer

Instruction Set

- Understanding the language of the hardware is key to understanding the hardware/software interface
- A program (in say, C) is compiled into an executable that is composed of machine instructions – this executable must also run on future machines – for example, each Intel processor reads in the same x86 instructions, but each processor handles instructions differently
- Java programs are converted into portable bytecode that is converted into machine instructions during execution (just-in-time compilation)
- What are important design principles when defining the instruction set architecture (ISA)?
  - ✓ keep the hardware simple – the chip must only implement basic primitives and run fast
  - ✓ keep the instructions regular – simplifies the decoding/scheduling of instructions

A Basic MIPS* Instruction

C code: a = b + c ;

Assembly code: (human-friendly machine instructions)

```
add  a, b, c     # a is the sum of b and c
```

Machine code: (hardware-friendly machine instructions)

00000010001100100100000000100000

Translate the C code into assembly code: a = b + c + d + e;

```
add  a, b, c          add  a, b, c
add  a, a, d   or     add  f, d, e
add  a, a, e          add  a, a, f
```

Note:

- Instructions are simple: fixed number of operands (unlike C)
- A single line of C code is converted into multiple lines of assembly code
- Some sequences are better than others. The second sequence needs one more (temporary) variable f

***MIPS**: Microprocessor Without Interlocked Pipelined Stages

Subtract Example

C code: f = (g + h) – (i + j);

translates into the following assembly code:

```
add  t0, g, h          add  f, g, h
add  t1,  i, j   or    sub  f, f, i
sub  f,  t0, t1        sub  f, f, j
```

Note: Each version may produce a different result because floating-point operations are not necessarily associative and commutative… more on this later

- Add and subtract, three operands: Two sources and one destination
- All arithmetic operations have this form

Design Principle 1: Simplicity favors regularity

✓ Regularity makes implementation simpler

✓ Simplicity enables higher performance at lower cost

Variables and Registers

- In C, each "variable" is a location in memory
- In hardware, each memory access is expensive
  - if a variable is accessed repeatedly, bring the variable into an on-chip scratchpad and operate on the scratchpad (registers)
- To simplify the instructions, we require that each instruction (add, sub) only operate on registers

Design Principle 2*:* Smaller is faster
c.f. main memory: millions of locations

- C code:

  f = (g + h) - (i + j);
  - f, …, j in $s0, …, $s4
- Compiled MIPS code:

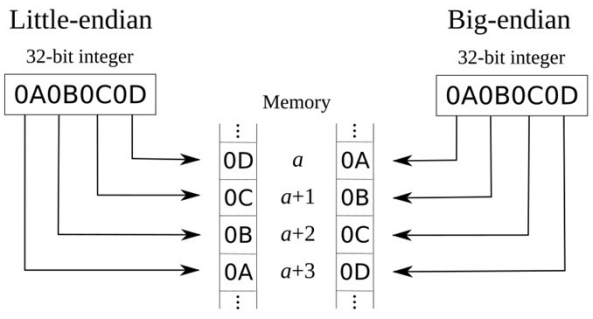
```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

- The MIPS ISA has 32 registers (x86 has 8 registers) – Why not more? Why not less?
- Each register is 32 bits wide (modern 64-bit architectures have 64-bit wide registers)
- A 32-bit entity (4 bytes) is referred to as a word
- To make the code more readable, registers are partitioned as $s0-$s7 (C/Java variables), $t0-$t9 (temporary variables)…

Note: the number of operands (variables) in a C program is very large; the number of operands in assembly is fixed.

Memory Operands

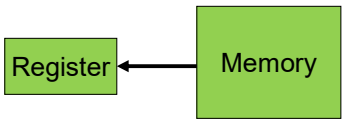
- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address



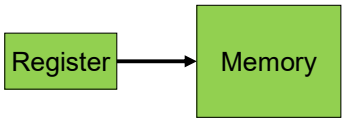
Memory Operands

- Values must be fetched from memory before (add and sub) instructions can operate on them

Load word
lw $t0, memory-address

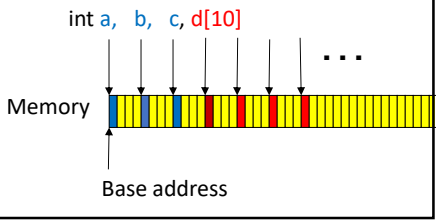


Store word
sw $t0, memory-address



How is memory-address determined?

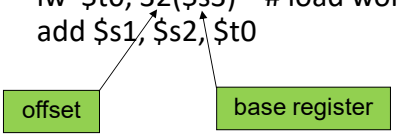
The compiler organizes data in memory…
It knows the location of every variable (saved in a table)…
It can fill in the appropriate mem-address for load-store instructions



Memory Operands: Example

- **C code:** g = h + A[8];
  - g in $s1, h in $s2, base address of A in $s3
- **Compiled MIPS code:**
  - Index 8 requires offset of 32 (4 bytes per word)

```
lw  $t0, 32($s3)   # load word
add $s1, $s2, $t0
```



- **C code:** A[12] = h + A[8];
  - h in $s2, base address of A in $s3

Compiled MIPS code:

```
lw  $t0, 32($s3)   # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)   # store word
```

Representing Instructions

MIPS R-format instructions

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

Register numbers

$t0 – $t7 are reg's 8 – 15
$t8 – $t9 are reg's 24 – 25
$s0 – $s7 are reg's 16 – 23

Instruction fields

- op: operation code (opcode)
- rs: first source register number
- rt: second source register number
- rd: destination register number
- shamt: shift amount (00000 for now)
- funct: function code (extends opcode)

Example: add $t0, $s1, $s2

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$0000\ 0010\ 0011\ 0010\ 0100\ 0000\ 0010\ 0000_2$
$= 02324020_{16}$

Note: for sub $t0, $s1, $s2 -- funct = 34

Representing Instructions

MIPS I-format instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

Register numbers

$t0 – $t7 are reg's 8 – 15
$t8 – $t9 are reg's 24 – 25
$s0 – $s7 are reg's 16 – 23

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs
- **Design Principle 4:** Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible
  - Formats are distinguished by op field

Example: lw $t0, 32($t1)

35	9	8	32

sw $t0, 32($t1)

43	9	8	32

addi $s0, $s0, 4

8	16	16	4

- No subi instruction
  Use addi $s0, $s0, -1

Shift Operations

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

- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - sll by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - srl by *i* bits divides by 2^i (unsigned only)

srl $s0, $t1, 10

0	0	9	16	10	2

Register numbers

$t0 – $t7 are reg's 8 – 15
$t8 – $t9 are reg's 24 – 25
$s0 – $s7 are reg's 16 – 23

s0 = 0000 0000 0000 0000 0000 0000 0000 1101

sll $t2, $s0, 8

0	0	16	8	8	0

t2 = 0000 0000 0000 0000 0000 1101 0000 0000

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- beq rs, rt, L1
  - if (rs == rt) branch to instruction labeled L1;
- bne rs, rt, L1
  - if (rs != rt) branch to instruction labeled L1;

```
Loop:sll  $t1,$s3,2
     add  $t1,$t1,$s6
     lw   $t0,0($t1)
     bne  $t0,$s5, Exit
     addi $s3,$s3,1
     j    Loop
Exit:
```

Register numbers

$t0 – $t7 are reg's 8 – 15
$t8 – $t9 are reg's 24 – 25
$s0 – $s7 are reg's 16 – 23

Example: beq $t0, $s0, 100

4	8	16	25

16

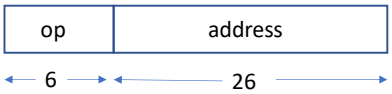
Address: PC + 4 + (25×4)

Note: For bne op = 5

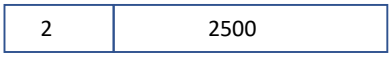
Unconditional Jumps

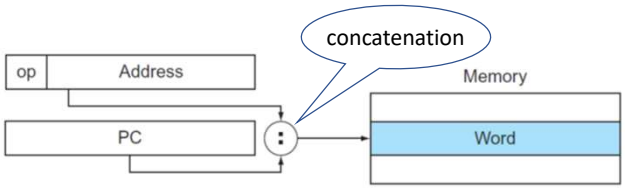
- j L1
  - unconditional jump to instruction labeled L1
- jr reg

J type Instruction



Example: J 10000



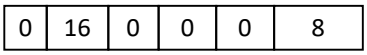


Target address = $PC_{31...28}$: (address × 4)

Register numbers

$t0 – $t7 are reg's 8 – 15
$t8 – $t9 are reg's 24 – 25
$s0 – $s7 are reg's 16 – 23

Example: jr $s0



Q. How to load a 32-bit constant into a reg?

lui $t0, 255 // Load upper immediate

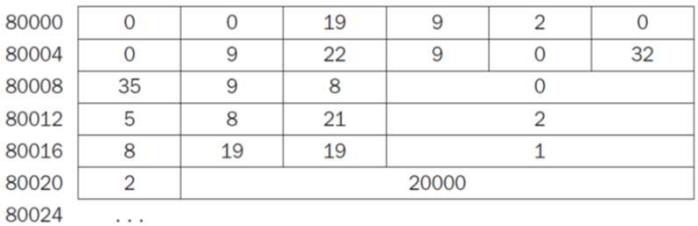
Example

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- beq rs, rt, L1 -- if (rs == rt) branch to instruction labeled L1;
- bne rs, rt, L1 -- if (rs != rt) branch to instruction labeled L1;
- j L1 -- unconditional jump to instruction labeled L1

C code: while (save[i] == k) i += 1;

- i in $s3, k in $s5, address of save in $s6

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```



C code: f = (i ==j)? g + h : g – h;

- f, g, … in $s0, $s1, …

Compiled MIPS code:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- slt rd, rs, rt
  - if (rs < rt) rd = 1; else rd = 0;
- slti rt, rs, constant
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
slt $t0, $s1, $s2    # if ($s1 < $s2)
bne $t0, $zero, L  # branch to L
```

- Signed comparison: slt, slti
- Unsigned comparison: sltu, sltui

- Why not blt, bge, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- beq and bne are the common case
- This is a good design compromise
- Can we design blt, bge using slt and beq/bne?

Example

$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
$s1 = 0000 0000 0000 0000 0000 0000 0000 0001

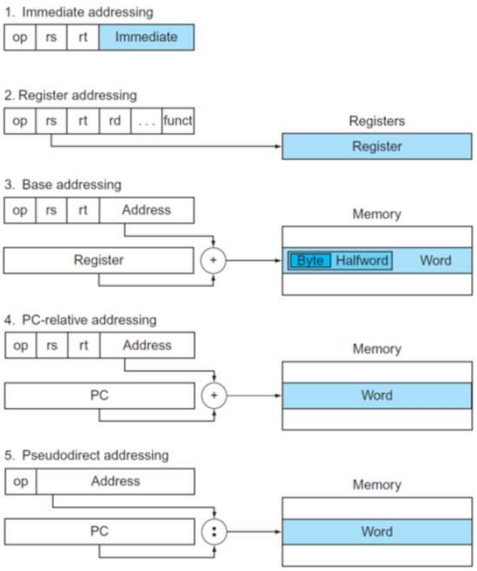
```
slt  $t0, $s0, $s1  # signed
```

–1 < +1 ⇒ $t0 = 1

```
sltu $t0, $s0, $s1  # unsigned
```

+4,294,967,295 > +1 ⇒ $t0 = 0

MIPS Addressing Mode



operand is a constant within the instruction

operand is a register

operand is at the memory location whose address is the sum of a register and a constant in the instruction

branch address is the sum of the PC and a constant in the instruction

jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

Procedure

Steps required

1. Place parameters in registers
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure's operations
5. Place result in register for caller
6. Return to place of call

Procedure call: jump and link

jal ProcedureLabel

- Address of following instruction put in $ra
  i.e., $ra = PC + 4
- Jumps to target address

Example: jal 10000

3	2500
6	26

Target address = $PC_{31\ldots28}$: (address × 4)

Procedure return: jump register

jr $ra

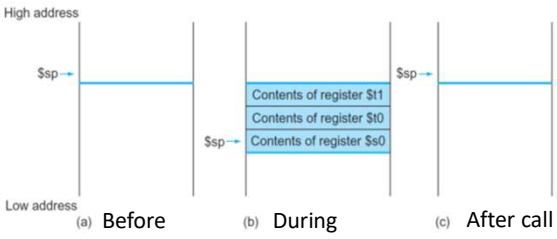
- Copies $ra to program counter
- Can also be used for computed jumps
  e.g., for case/switch statements

Leaf Procedure Example

- C code:

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

- Arguments g, …, j in $a0, …, $a3
- f in $s0
- Result in $v0



leaf_example:

```
addi $sp, $sp, -12  # adjust stack to make room for 3 items
sw   $t1, 8($sp)    # save register $t1 for use afterwards
sw   $t0, 4($sp)    # save register $t0 for use afterwards
sw   $s0, 0($sp)    # save register $s0 for use afterwards

add $t0,$a0,$a1 # register  $t0 contains g + h
add $t1,$a2,$a3 # register  $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, which is (g + h)-(i + j)

add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)

lw $s0, 0($sp)  # restore register $s0 for caller
lw $t0, 4($sp)  # restore register $t0 for caller
lw $t1, 8($sp)  # restore register $t1 for caller
addi $sp,$sp,12 # adjust stack to delete 3 items

jr   $ra    # jump back to calling routine
```

Non-leaf Procedure

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

C code:

```
int rec_sum (int n)
{
 if (n < 1) return 1;
 else return n + rec_sum (n - 1);
}
```

Register usage convention by MIPS software

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

Non-leaf Procedure Example

C code:

```
int rec_sum (int n)
{
 if (n < 1) return 1;
 else return n + rec_sum (n - 1);
}
```

- Argument n in $a0
- Result in $v0

MIPS code (follows register usage convention):

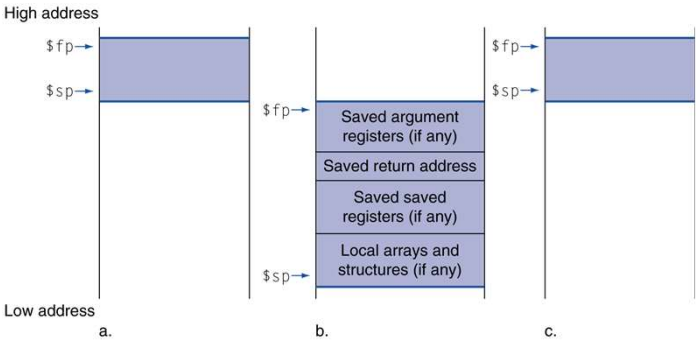
```
rec_sum:
     slti $t0, $a0, 1      # test for n < 1
     beq  $t0, $zero, L1
     addi $v0, $zero, 1    # if so, result is 1
     jr   $ra              #   and return

L1:  addi $sp, $sp, -8     # adjust stack for 2 items
     sw   $ra, 4($sp)      # save return address
     sw   $a0, 0($sp)      # save argument
     addi $a0, $a0, -1     # else decrement n
     jal rec_sum           # recursive call

     lw   $a0, 0($sp)      # restore original n
     lw   $ra, 4($sp)      #   and return address
     addi $sp, $sp, 8      # pop 2 items from stack

     add  $v0, $a0, $v0    # add to get result
     jr   $ra              # and return
```

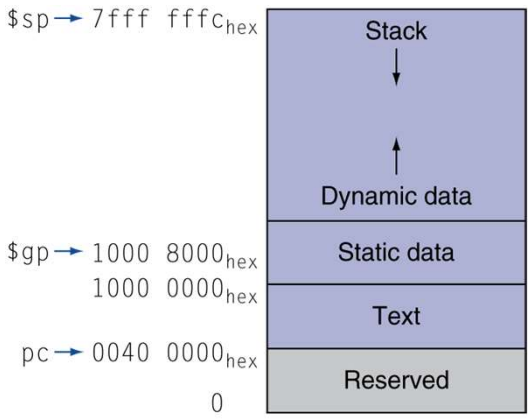
Local Data on the Stack



- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage
  - Frame pointer ($fp) points to the first word of the frame of a procedure
  - Frame pointer offers a stable base register within a procedure for local memory references

Memory Layout

- **Text:** program code
- **Static data:** global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- **Dynamic data:** heap
  - E.g., malloc in C
- **Stack:** automatic storage



Arrays Vs Pointers

```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

Note: $a0 = array, $a1 = size

```
    move $t0,$zero   # i = 0
L1: sll $t1,$t0,2    # $t1 = i * 4
    add $t2,$a0,$t1  # $t2 = &array[i]
    sw $zero, 0($t2) # array[i] = 0
    addi $t0,$t0,1   # i = i + 1
    slt $t3,$t0,$a1  # $t3 = (i < size)
    bne $t3,$zero,L1 # if (…) goto L1
```

```
    move $t0,$a0    # p = &array[0]
    sll $t1,$a1,2   # $t1 = size * 4
    add $t2,$a0,$t1 # $t2 = array[size]
L2: sw $zero,0($t0) # Memory[p] = 0
    addi $t0,$t0,4  # p = p + 4
    slt $t3,$t0,$t2 # $t3=(p < &array[size])
    bne $t3,$zero,L2 # if (…) goto L2
```

Concluding Remarks

- **Design principles**
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86