Signed vs. Unsigned

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

Example

- `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
- `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
- `slt $t0, $s0, $s1` # signed
  - $-1 < +1$ $\Rightarrow$ $t0 = 1$
- `sltu $t0, $s0, $s1` # unsigned
  - $+4,294,967,295 > +1$ $\Rightarrow$ $t0 = 0$
Procedure Calling

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
Register Usage

- $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)
Procedure Call Instructions

- Procedure call: jump and link
  \texttt{jal ProcedureLabel}
  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: jump register
  \texttt{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:

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

- Arguments g, ..., j in $a0, ..., $a3
- f in $s0 (hence, need to save $s0 on stack)
- Result in $v0
### Leaf Procedure Example

**MIPS code:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $sp, $sp, -4</td>
<td>Save $s0 on stack</td>
</tr>
<tr>
<td>sw $s0, 0($sp)</td>
<td>Procedure body</td>
</tr>
<tr>
<td>add $t0, $a0, $a1</td>
<td></td>
</tr>
<tr>
<td>add $t1, $a2, $a3</td>
<td></td>
</tr>
<tr>
<td>sub $s0, $t0, $t1</td>
<td>Result</td>
</tr>
<tr>
<td>add $v0, $s0, $zero</td>
<td></td>
</tr>
<tr>
<td>lw $s0, 0($sp)</td>
<td></td>
</tr>
<tr>
<td>addi $sp, $sp, 4</td>
<td>Restore $s0</td>
</tr>
<tr>
<td>jr $ra</td>
<td>Return</td>
</tr>
</tbody>
</table>
Non-Leaf Procedures

- 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
Non-Leaf Procedure Example

- C code:
  ```c
  int fact (int n)
  {
    if (n < 1) return f;
    else return n * fact(n - 1);
  }
  ```
- Argument n in $a0
- Result in $v0
Non-Leaf Procedure Example

- MIPS code:

<table>
<thead>
<tr>
<th>fact:</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $sp, $sp, -8  # adjust stack for 2 items</td>
</tr>
<tr>
<td>sw $ra, 4($sp)    # save return address</td>
</tr>
<tr>
<td>sw $a0, 0($sp)    # save argument</td>
</tr>
<tr>
<td>slti $t0, $a0, 1  # test for n &lt; 1</td>
</tr>
<tr>
<td>beq $t0, $zero, L1</td>
</tr>
<tr>
<td>addi $v0, $zero, 1 # if so, result is 1</td>
</tr>
<tr>
<td>addi $sp, $sp, 8  # pop 2 items from stack</td>
</tr>
<tr>
<td>jr $ra             # and return</td>
</tr>
<tr>
<td>L1: addi $a0, $a0, -1 # else decrement n</td>
</tr>
<tr>
<td>jal fact          # recursive call</td>
</tr>
<tr>
<td>lw $a0, 0($sp)    # restore original n</td>
</tr>
<tr>
<td>lw $ra, 4($sp)    # and return address</td>
</tr>
<tr>
<td>addi $sp, $sp, 8  # pop 2 items from stack</td>
</tr>
<tr>
<td>mul $v0, $a0, $v0 # multiply to get result</td>
</tr>
<tr>
<td>jr $ra            # and return</td>
</tr>
</tbody>
</table>
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
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, new in Java
- Stack: automatic storage
Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, ...
  - Most of the world’s alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings
Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case
    - `lb rt, offset(rs)`  `lh rt, offset(rs)`
    - Sign extend to 32 bits in rt
      - `lbu rt, offset(rs)`  `lhu rt, offset(rs)`
    - Zero extend to 32 bits in rt
      - `sb rt, offset(rs)`  `sh rt, offset(rs)`
  - Store just rightmost byte/halfword
String Copy Example

- C code (naïve):
  - Null-terminated string
  
  ```c
  void strcpy (char x[], char y[])
  {
    int i;
    i = 0;
    while ((x[i] = y[i]) != '\0')
      i += 1;
  }
  ```
  
  - Addresses of x, y in $a0, $a1
  - i in $s0
String Copy Example

- **MIPS code:**

<table>
<thead>
<tr>
<th>Assembly Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $sp, $sp, -4</td>
<td># adjust stack for 1 item</td>
</tr>
<tr>
<td>sw $s0, 0($sp)</td>
<td># save $s0</td>
</tr>
<tr>
<td>add $s0, $zero, $zero # i = 0</td>
<td></td>
</tr>
<tr>
<td>L1: add $t1, $s0, $a1 # addr of y[i] in $t1</td>
<td></td>
</tr>
<tr>
<td>lbu $t2, 0($t1)      # $t2 = y[i]</td>
<td></td>
</tr>
<tr>
<td>add $t3, $s0, $a0    # addr of x[i] in $t3</td>
<td></td>
</tr>
<tr>
<td>sb $t2, 0($t3)       # x[i] = y[i]</td>
<td></td>
</tr>
<tr>
<td>beq $t2, $zero, L2   # exit loop if y[i] == 0</td>
<td></td>
</tr>
<tr>
<td>addi $s0, $s0, 1     # i = i + 1</td>
<td></td>
</tr>
<tr>
<td>j L1                  # next iteration of loop</td>
<td></td>
</tr>
<tr>
<td>L2: lw $s0, 0($sp)   # restore saved $s0</td>
<td></td>
</tr>
<tr>
<td>addi $sp, $sp, 4     # pop 1 item from stack</td>
<td></td>
</tr>
<tr>
<td>jr $ra                # and return</td>
<td></td>
</tr>
</tbody>
</table>
32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
  lui rt, constant
  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

```
lhi $s0, 61 0000 0000 0111 1101 0000 0000 0000 0000
ori $s0, $s0, 2304 0000 0000 0111 1101 0000 1001 0000 0000
```

Chapter 2 — Instructions: Language of the Computer — 51
Branch Addressing

Branch instructions specify
- Opcode, two registers, target address

Most branch targets are near branch
- Forward or backward

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>constant or address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

PC-relative addressing
- Target address = PC + offset × 4
- PC already incremented by 4 by this time
Jump Addressing

- Jump (j and jal) targets could be anywhere in text segment
  - Encode full address in instruction

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

- (Pseudo)Direct jump addressing
  - Target address = PC_{31...28} : (address \times 4)
Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

<table>
<thead>
<tr>
<th>Loop: sll $t1, $s3, 2</th>
<th>80000</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $t1, $t1, $s6</td>
<td>80004</td>
</tr>
<tr>
<td>lw $t0, 0($t1)</td>
<td>80008</td>
</tr>
<tr>
<td>bne $t0, $s5, Exit</td>
<td>80012</td>
</tr>
<tr>
<td>addi $s3, $s3, 1</td>
<td>80016</td>
</tr>
<tr>
<td>j Loop</td>
<td>80020</td>
</tr>
<tr>
<td>Exit: ...</td>
<td>80024</td>
</tr>
</tbody>
</table>
Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code.

Example

```
beq $s0,$s1, L1
↓
bne $s0,$s1, L2
    j L1
L2: ...
```
Addressing Mode Summary

1. Immediate addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Immediate} \]

2. Register addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{rd} \quad \ldots \quad \text{func} \]
   \[ \text{Registers} \quad \text{Register} \]

3. Base addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Address} \]
   \[ \text{Register} \quad + \quad \text{Byte} \quad \text{Halfword} \quad \text{Word} \]

4. PC-relative addressing
   \[ \text{op} \quad \text{rs} \quad \text{rt} \quad \text{Address} \]
   \[ \text{PC} \quad + \quad \text{Word} \]

5. Pseudodirect addressing
   \[ \text{op} \quad \text{Address} \]
   \[ \text{PC} \quad + \quad \text{Word} \]
Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don’t synchronize
    - Result depends on order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions
Synchronization in MIPS

- **Load linked:** `ll rt, offset(rs)`
- **Store conditional:** `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in `rt`
  - Fails if location is changed
    - Returns 0 in `rt`
- **Example: atomic swap (to test/set lock variable)**
  ```
  try: add $t0,$zero,$s4 ;copy exchange value
     ll $t1,0($s1)    ;load linked
     sc $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
  add $s4,$zero,$t1 ;put load value in $s4
  ```
Many compilers produce object modules directly.

Static linking
Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler’s imagination

move $t0, $t1  →  add $t0, $zero, $t1
blt $t0, $t1, L  →  slt $at, $t0, $t1
                         bne $at, $zero, L

- $at (register 1): assembler temporary
Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code
Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs

- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space
Loading a Program

Load from image file on disk into memory

1. Read header to determine segment sizes
2. Create virtual address space
3. Copy text and initialized data into memory
   - Or set page table entries so they can be faulted in
4. Set up arguments on stack
5. Initialize registers (including $sp, $fp, $gp)
6. Jump to startup routine
   - Copies arguments to $a0, … and calls main
   - When main returns, do exit syscall
Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions
Lazy Linkage

Indirection table

Stub: Loads routine ID, Jump to linker/loader

Linker/loader code

Dynamically mapped code

a. First call to DLL routine

b. Subsequent calls to DLL routine