MIPS Instruction Set Architecture (2)

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Outline

Textbook: P&H 2.5-2.12 (Except 2.11) – for 4th Ed.
    P&H 2.5-2.11 (Except 2.10) – for 5th Ed.

• Representing Instructions
• Logical Operations
• Instructions for Making Decisions
• Supporting Procedures in Computer Hardware
• Communicating with People
• MIPS Addressing for 32-Bit Immediates and Addresses
• Translating and Starting a Program
• Array vs. Pointers – Will not cover (2.14 in 4th Ed.)
Representing Instructions
Representing Instructions

- Instructions are encoded in binary
  - Called machine code

- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

- Register numbers
  - $t0 - t7$ are reg’s 8 – 15
  - $t8 - t9$ are reg’s 24 – 25
  - $s0 - s7$ are reg’s 16 – 23
MIPS R-format Instructions

- **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)
**R-format Example**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

**add $t0, $s1, $s2**

<table>
<thead>
<tr>
<th>special</th>
<th>$s1</th>
<th>$s2</th>
<th>$t0</th>
<th>0</th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
<td>18</td>
<td>8</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
</tr>
</tbody>
</table>

00000010 00110010 01000000 00100000₂ = 02324020₁₆
MIPS I-format Instructions

<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>

- 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
MIPS J-format Instructions

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

- Jump instructions (j and jal)
  - Address: encode 26-bit target address
Stored Program Computers

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs
Logical Operations
Logical Operations

• **Instructions for bitwise manipulation**

<table>
<thead>
<tr>
<th>Operation</th>
<th>C</th>
<th>Java</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left</td>
<td>&lt;&lt;</td>
<td>&lt;&lt;</td>
<td>sll</td>
</tr>
<tr>
<td>Shift right</td>
<td>&gt;&gt;</td>
<td>&gt;&gt;</td>
<td>srl</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>&amp;</td>
<td>and, andi</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td></td>
<td></td>
<td>or, ori</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td>~</td>
<td>~</td>
<td>nor</td>
</tr>
</tbody>
</table>

Useful for extracting and inserting groups of bits in a word
Shift Operations

- **shamt**: how many positions to shift
- **Shift left logical**
  - Shift left and fill with 0 bits
  - \texttt{sll} by \( i \) bits multiplies by \( 2^i \)
- **Shift right logical**
  - Shift right and fill with 0 bits
  - \texttt{srl} by \( i \) bits divides by \( 2^i \) (unsigned only)
  - cf. \texttt{sra}: shift right arithmetic
AND Operations

• Useful to mask bits in a word
  – Select some bits, clear others to 0

and $t0$, $t1$, $t2$

<table>
<thead>
<tr>
<th>$t2$</th>
<th>0000 0000 0000 0000 0000 1101 1100 0000</th>
<th>(data)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t1$</td>
<td>0000 0000 0000 0000 0011 1100 0000 0000</td>
<td>(mask)</td>
</tr>
<tr>
<td>$t0$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

or $t0$, $t1$, $t2$

\[
\begin{array}{c|c}
\text{data} & 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 1101 \ 1100 \ 0000 \\
\text{mask}  & 0000 \ 0000 \ 0000 \ 0000 \ 0011 \ 1100 \ 0000 \ 0000 \\
\end{array}
\]
NOT Operations

• Useful to invert bits in a word
  – Change 0 to 1, and 1 to 0
• MIPS has NOR 3-operand instruction
  – a NOR b == NOT ( a OR b )

```plaintext
nor $t0, $t1, $zero
```

<table>
<thead>
<tr>
<th>$t1</th>
<th>0000 0000 0000 0000 0011 1100 0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t0</td>
<td>1111 1111 1111 1111 1100 0011 1111 1111</td>
</tr>
</tbody>
</table>

Register 0: always read as zero
Instructions for Making Decisions
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;
• j L1
  – unconditional jump to instruction labeled L1
Compiling If Statements

• C code:
  
  ```
  if (i == j) f = g+h;
  else f = g-h;
  ```
  
  - f, g, ... in $s0, $s1, ... 

• Compiled MIPS code:
Compiling Loop Statements

• C code:
  
  ```
  while (save[i] == k) i += 1;
  ```
  
  – i in $s3, k in $s5, address of save in $s6

• Compiled MIPS code:
  
  ```
  Loop:  sll  $t1, $s3, 2
         add  $t1, $t1, $s6
         lw    $t0, 0($t1)
         bne  $t0, $s5, Exit
         addi $s3, $s3, 1
         j    Loop
  
  Exit: …
  ```
Basic Blocks

• A basic block is a sequence of instructions with
  – No embedded branches (except at end)
  – No branch targets (except at beginning)

• A compiler identifies basic blocks for optimization

• An advanced processor can accelerate execution of basic blocks
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
  ```
Branch Instruction Design

• 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
Signed vs. Unsigned

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

\[
\begin{align*}
  s0 &= 1 \ldots 1 \ldots 1 \ldots 1 \\
  s1 &= 0 \ldots 0 \ldots 0 \ldots 0 \\
  - \text{slt } t0, s0, s1 & \quad \# \text{ signed} \\
  -1 < +1 & \Rightarrow t0 = 1 \\
  - \text{sltu } t0, s0, s1 & \quad \# \text{ unsigned} \\
  +4,294,967,295 > +1 & \Rightarrow t0 = 0
\end{align*}
\]
Supporting Procedures in Computer Hardware
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 (caller-save reg’s)
  – Can be overwritten by callee
• $s0 – $s7: saved (callee-saved reg’s)
  – 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 (Continued)

- **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></td>
</tr>
<tr>
<td>add  $v0, $s0, $zero</td>
<td>Result</td>
</tr>
<tr>
<td>lw   $s0, 0($sp)</td>
<td>Restore $s0</td>
</tr>
<tr>
<td>addi $sp, $sp, 4</td>
<td>Return</td>
</tr>
<tr>
<td>jr    $ra</td>
<td></td>
</tr>
</tbody>
</table>
```

Leaf_example:
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>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $sp, $sp, -8</td>
<td># adjust stack for 2 items</td>
</tr>
<tr>
<td>sw $ra, 4($sp)</td>
<td># save return address</td>
</tr>
<tr>
<td>sw $a0, 0($sp)</td>
<td># save argument</td>
</tr>
<tr>
<td>slti $t0, $a0, 1</td>
<td># test for n &lt; 1</td>
</tr>
<tr>
<td>beq $t0, $zero, L1</td>
<td></td>
</tr>
</tbody>
</table>
```

```
addi $v0, $zero, 1    # if so, result is 1
addi $sp, $sp, 8      # pop 2 items from stack
jr $ra                # and return

L1: addi $a0, $a0, -1  # else decrement n
jal fact               # 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
mul $v0, $a0, $v0      # multiply to get result
jr $ra                 # and return
```
### fact() Animation:

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
</tr>
<tr>
<td>0x0FC</td>
</tr>
<tr>
<td>0x0F8</td>
</tr>
<tr>
<td>0x0F4</td>
</tr>
<tr>
<td>0x0F0</td>
</tr>
<tr>
<td>0x0EC</td>
</tr>
<tr>
<td>0x0E8</td>
</tr>
<tr>
<td>0x0E4</td>
</tr>
<tr>
<td>0x0E0</td>
</tr>
<tr>
<td>0x0DC</td>
</tr>
<tr>
<td>0x0D8</td>
</tr>
<tr>
<td>0x0D4</td>
</tr>
<tr>
<td>0x0D0</td>
</tr>
<tr>
<td>0x0CC</td>
</tr>
<tr>
<td>0x0C8</td>
</tr>
<tr>
<td>0x0C4</td>
</tr>
</tbody>
</table>

- **main's fp**
- **main's sp (0x100)**
### fact() Animation:

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td></td>
</tr>
<tr>
<td>0x0FC</td>
<td>main's return address</td>
</tr>
<tr>
<td>0x0F8</td>
<td>space for $a0</td>
</tr>
<tr>
<td>0x0F4</td>
<td></td>
</tr>
<tr>
<td>0x0F0</td>
<td></td>
</tr>
<tr>
<td>0x0EC</td>
<td></td>
</tr>
<tr>
<td>0x0E8</td>
<td></td>
</tr>
<tr>
<td>0x0E4</td>
<td></td>
</tr>
<tr>
<td>0x0E0</td>
<td></td>
</tr>
<tr>
<td>0x0DC</td>
<td></td>
</tr>
<tr>
<td>0x0D8</td>
<td></td>
</tr>
<tr>
<td>0x0D4</td>
<td></td>
</tr>
<tr>
<td>0x0D0</td>
<td></td>
</tr>
<tr>
<td>0x0CC</td>
<td></td>
</tr>
<tr>
<td>0x0C8</td>
<td></td>
</tr>
<tr>
<td>0x0C4</td>
<td></td>
</tr>
</tbody>
</table>

Frame for fact(10)

fact(10)'s fp

fact(10)'s sp
**fact() Animation:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td></td>
</tr>
<tr>
<td>0x0FC</td>
<td>main's return address</td>
</tr>
<tr>
<td>0x0F8</td>
<td>space for $a0 = 10</td>
</tr>
<tr>
<td>0x0F4</td>
<td>fact(10)'s return address</td>
</tr>
<tr>
<td>0x0F0</td>
<td>space for $a0 = 9</td>
</tr>
<tr>
<td>0x0EC</td>
<td></td>
</tr>
<tr>
<td>0x0E8</td>
<td></td>
</tr>
<tr>
<td>0x0E4</td>
<td></td>
</tr>
<tr>
<td>0x0E0</td>
<td></td>
</tr>
<tr>
<td>0x0DC</td>
<td></td>
</tr>
<tr>
<td>0x0D8</td>
<td></td>
</tr>
<tr>
<td>0x0D4</td>
<td></td>
</tr>
<tr>
<td>0x0D0</td>
<td></td>
</tr>
<tr>
<td>0x0CC</td>
<td></td>
</tr>
<tr>
<td>0x0C8</td>
<td></td>
</tr>
<tr>
<td>0x0C4</td>
<td></td>
</tr>
</tbody>
</table>

- Frame for fact(10)
- Frame for fact(9)

Stack grows in this direction
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
Communicating with People
(Handling Characters)
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
    
    ```
    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:**

```mips
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw $s0, 0($sp)          # save $s0
    add $s0, $zero, $zero   # i = 0

L1:   add $t1, $s0, $a1    # addr of y[i] in $t1
    lbu $t2, 0($t1)        # $t2 = y[i]
    add $t3, $s0, $a0      # addr of x[i] in $t3
    sb $t2, 0($t3)         # x[i] = y[i]
    beq $t2, $zero, L2     # exit loop if y[i] == 0
    addi $s0, $s0, 1       # i = i + 1
    j L1                    # next iteration of loop

L2:   lw $s0, 0($sp)       # restore saved $s0
    addi $sp, $sp, 4       # pop 1 item from stack
    jr $ra                  # and return
```
MIPS Addressing for 32-Bit Immediates and Addresses
32-bit Constants

• Most constants are small
  – 16-bit immediate is sufficient

• For the occasional 32-bit constant
  
  \texttt{lui \ rt, constant}
  – Copies 16-bit constant to left 16 bits of \rt
  – Clears right 16 bits of \rt to 0

\texttt{lhi \ $s0, 61}

\texttt{ori \ $s0, \ $s0, 2304}
Branch Addressing

• Branch instructions specify
  – Opcode, two registers, target address

<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>

• Most branch targets are near branch
  – Forward or backward

• 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\ldots28} : (address \times 4)
Target Addressing Example

• Loop code from earlier example
  – Assume Loop at location 80000

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>20000</th>
<th>19</th>
<th>9</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll $t1, $s3, 2</td>
<td>80000</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>add $t1, $t1, $s6</td>
<td>80004</td>
<td>0</td>
<td>9</td>
<td>22</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>lw $t0, 0($t1)</td>
<td>80008</td>
<td>35</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>bne $t0, $s5, Exit</td>
<td>80012</td>
<td>5</td>
<td>8</td>
<td>21</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>addi $s3, $s3, 1</td>
<td>80016</td>
<td>8</td>
<td>19</td>
<td>19</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j Loop</td>
<td>80020</td>
<td>2</td>
<td></td>
<td>19</td>
<td>20000</td>
<td></td>
</tr>
<tr>
<td>Exit: …</td>
<td>80024</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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
   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} \\
   \text{Immediate}
   \end{array}
   \]

2. Register addressing
   \[
   \begin{array}{cccc}
   \text{op} & \text{rs} & \text{rt} & \text{rd} & \ldots & \text{func}
   \end{array}
   \]

3. Base addressing
   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} \\
   \text{Address}
   \end{array}
   \]

4. PC-relative addressing
   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} \\
   \text{Address}
   \end{array}
   \]

5. Pseudodirect addressing
   \[
   \begin{array}{c}
   \text{op} \\
   \text{Address}
   \end{array}
   \]

   \[
   \begin{array}{c}
   \text{PC} \\
   \vdots
   \end{array}
   \]

   \[
   \begin{array}{c}
   \text{Word}
   \end{array}
   \]
Translating and Starting a Program
Translation and Startup

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
Starting Java Applications

1. Java program
2. Class files (Java bytecodes)
3. Compiled Java methods (machine language)
4. Java library routines (machine language)
5. Interprets bytecodes
6. Just In Time compiler
7. Java Virtual Machine
8. Simple portable instruction set for the JVM

Compiles bytecodes of “hot” methods into native code for host machine
Array vs. Pointers
Arrays vs. Pointers

• **Array indexing involves**
  – Multiplying index by element size
  – Adding to array base address

• **Pointers correspond directly to memory addresses**
  – Can avoid indexing complexity
### Example: Clearing and Array

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

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

#### Assembly Code

<table>
<thead>
<tr>
<th>clear1</th>
<th>clear2</th>
</tr>
</thead>
<tbody>
<tr>
<td>move $t0,$zero  # i = 0</td>
<td>move $t0,$a0  # p = &amp; array[0]</td>
</tr>
<tr>
<td>loop1: sll $t1,$t0,2  # $t1 = i * 4</td>
<td>sll $t1,$a1,2  # $t1 = size * 4</td>
</tr>
<tr>
<td>add $t2,$a0,$t1  # $t2 =</td>
<td>add $t2,$a0,$t1  # $t2 =</td>
</tr>
<tr>
<td># &amp;array[i]</td>
<td># &amp;array[size]</td>
</tr>
<tr>
<td>sw $zero, 0($t2) # array[i] = 0</td>
<td>sw $zero,0($t0) # Memory[p] = 0</td>
</tr>
<tr>
<td>addi $t0,$t0,1  # i = i + 1</td>
<td>addi $t0,$t0,4  # p = p + 4</td>
</tr>
<tr>
<td>slt $t3,$t0,$a1  # $t3 =</td>
<td>slt $t3,$t0,$t2  # $t3 =</td>
</tr>
<tr>
<td># (i &lt; size)</td>
<td># (p&lt;&amp;array[size])</td>
</tr>
<tr>
<td>bne $t3,$zero,loop1 # if (…</td>
<td>bne $t3,$zero,loop2 # if (…</td>
</tr>
<tr>
<td># goto loop1</td>
<td># goto loop2</td>
</tr>
</tbody>
</table>
Comparison of Array vs. Ptr

• Multiply “strength reduced” to shift
• Array version requires shift to be inside loop
  – Part of index calculation for incremented i
  – c.f. incrementing pointer
• Compiler can achieve same effect as manual use of pointers
  – Induction variable elimination
  – Better to make program clearer and safer