Instruction Set Architectures
Part I: From C to MIPS

Readings: 2.1 - 2.14
Goals for this Class

- Understand how CPUs run programs
  - How do we express the computation the CPU?
  - How does the CPU execute it?
  - How does the CPU support other system components (e.g., the OS)?
  - What techniques and technologies are involved and how do they work?
- Understand why CPU performance (and other metrics) varies
  - How does CPU design impact performance?
  - What trade-offs are involved in designing a CPU?
  - How can we meaningfully measure and compare computer systems?
- Understand why program performance varies
  - How do program characteristics affect performance?
  - How can we improve a program’s performance by considering the CPU running it?
  - How do other system components impact program performance?
Goals

• Understand how we express programs to the computer.
  • The stored-program model
  • The instruction set architecture

• Learn to read and write MIPS assembly

• Prepare for your 141L Project and 141 homeworks
  • Your book (and my slides) use MIPS throughout
  • You will implement a subset of MIPS in 141L

• Learn to “see past your code” to the ISA
  • Be able to look at a piece of C code and know what kinds of instructions it will produce.
  • Begin to understand the compiler’s role
  • Be able to roughly estimate the performance of code based on this understanding (we will refine this skill throughout the quarter.)
The Idea of the CPU
In the beginning...

- Physical configuration specified the computation a computer performed

The Difference Engine

ENIAC
TheStoredProgramComputer

• The program is *data*
  • It is a series of bits
  • It lives in memory
  • A series of discrete “instructions”

• The program counter (PC) control execution
  • It points to the current instruction
  • Advances through the program
The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program
The program is *data*
- It is a series of bits
- It lives in memory
- A series of discrete "instructions"

The program counter (PC) control execution
- It points to the current instruction
- Advances through the program
The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete "instructions"
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program
The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program
The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program

```
CPU
PC

Instruction Memory

[80000180] 0001d821  addu $27, $0, $1
[80000184] 30190000  lui $1, -28672
[80000188] ac220200  sw $2, 512($1)
[8000018c] 30190000  lui $1, -28672
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6000  mfc0 $26, $13
[80000198] 0012082  srl $4, $26, 2
[8000019c] 3084001f  and $4, $4, 31
[800001a0] 34020004  ori $2, $0, 4
[800001a4] 30049000  lui $4, -28672 [__m1__]
[800001a8] 0000000c  syscall
[800001ac] 34020001  ori $2, $0, 1
[800001b0] 0012082  srl $4, $26, 2
[800001b4] 3084001f  and $4, $4, 31

Data Memory

[7fff6e0] 74736574 73612e32 test 2.as
[7fff6e0] 75524553 54584554 $ERTTEXT
[7fff6e8] 78303d47 3a364631 G = 0x1F61
[7fff6e9] 5f444fe1 4554444d AND_MODE
[7fff6e0] 70410333 565f6073 3 Apple_
[7fff6e9] 6566b366 65525f74 packet_Re
[7fff6ec] 61c2f70 68636e6f_p/launch
```
The Stored Program Computer

• The program is *data*
  • It is a series of bits
  • It lives in memory
  • A series of discrete “instructions”

• The program counter (PC) control execution
  • It points to the current instruction
  • Advances through the program

```
CPU

PC

Instruction Memory
[80000180] 0001d821 addu $27, $0, $1
[80000184] 3c019000 lui $1, -28672
[80000188] ac220200 sw $2, 512($1)
[80000190] 3c019000 lui $1, -28672
[80000194] ac240204 sw $4, 516($1)
[80000198] 401a6800 mfco $26, $13
[8000019c] 001a2082 srl $4, $26, 2
[800001a0] 3084d001f andi $4, $4, 31
[800001a4] 3c049000 lui $4, -28672 (_m1_)
[800001a8] 0000000c syscall
[800001ac] 34020004 ori $2, $0, 4
[800001b0] 001a2082 srl $4, $26, 2
[800001b4] 3084d001f andi $4, $4, 31

Data Memory
[7fffe0] 74736574 test2.as
[7fffe0] 5f524553 54584554 54552574 Pocket_R
[7fffe0] 5f444e41 45544444 AND_MODE
[7fffe0] 70410033 3 Apple
[7fffe0] 65666366 6552574 Pocket_R
[7fffe0] 616c2f70 68636e75 p_launch
```
The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program
The Stored Program Computer

- The program is *data*
  - It is a series of bits
  - It lives in memory
  - A series of discrete “instructions”
- The program counter (PC) control execution
  - It points to the current instruction
  - Advances through the program

```
Instruction Memory
[80000180] 0001d821  addu $27, $0, $1
[80000184] 3c019000  lui $1, -28672
[80000188] ac220200  sw $2, $12($1)
[8000018c] 3c019000  lui $1, -28672
[80000190] ac240204  sw $4, $16($1)
[80000194] 401a6000  mfco $26, $13
[80000198] 001a2082  srl $4, $26, 2
[8000019c] 3084000f  andi $4, $4, 31
[800001a0] 34020004  ori $2, $0, 4
[800001a4] 3c049000  lui $4, -28672 [__m1__]
[800001a8] 00000000  syscall
[800001ac] 34020001  ori $2, $0, 1
[800001b0] 001a2082  srl $4, $26, 2
[800001b4] 3084000f  andi $4, $4, 31

Data Memory
[7fffe060] 74736574 73612e32 test2.asm
[7fffe070] 5f524553 54584554 test
[7fffe080] 78303d47 aaed4651 G-0*16
[7fffe090] 5f444441 4544444d AND_MODE
[7fffe0a0] 70420040 3 Apple_
[7fffe0b0] 65566366 65525744ocket_Re
[7fffe0c0] 616e2f70 68616e75 p/launch
```
The Instruction Set Architecture (ISA)

• The ISA is the set of instructions a computer can execute
• All programs are combinations of these instructions
• It is an abstraction that programmers (and compilers) use to express computations
  • The ISA defines a set of operations, their semantics, and rules for their use.
  • The software agrees to follow these rules.
• The hardware can implement those rules IN ANY WAY IT CHOOSES!
  • Directly in hardware
  • Via a software layer (i.e., a virtual machine)
  • Via a trained monkey with a pen and paper
  • Via a software simulator (like SPIM)
• Also called “the big A architecture”
The MIPS ISA
We Will Study Two ISAs

• MIPS
  • Simple, elegant, easy to implement
  • Designed with the benefit many years ISA design experience
  • Designed for modern programmers, tools, and applications
  • The basis for your implementation project in 141L
  • Not widely used in the real world (but similar ISAs are pretty common, e.g. ARM)

• x86
  • Ugly, messy, inelegant, crufty, arcane, very difficult to implement.
  • Designed for 1970s technology
  • Nearly the last in long series of unfortunate ISA designs.
  • The dominant ISA in modern computer systems.
We Will Study Two ISAs

• **MIPS**
  • Simple, elegant, easy to implement
  • Designed with the benefit many years ISA design experience
  • Designed for modern programmers, tools, and applications
  • The basis for your implementation project in 141L
  • Not widely used in the real world (but similar ISAs are pretty common, e.g. ARM)

• **x86**
  • Ugly, messy, inelegant, crufty, arcane, very difficult to implement.
  • Designed for 1970s technology
  • Nearly the last in long series of unfortunate ISA designs.
  • The dominant ISA in modern computer systems.

You will learn to write MIPS code and implement a MIPS processor.
We Will Study Two ISAs

- **MIPS**
  - Simple, elegant, easy to implement
  - Designed with the benefit many years ISA design experience
  - Designed for modern programmers, tools, and applications
  - The basis for your implementation project in 141L
  - Not widely used in the real world (but similar ISAs are pretty common, e.g. ARM)

- **x86**
  - Ugly, messy, inelegant, crufty, arcane, very difficult to implement.
  - Designed for 1970s technology
  - Nearly the last in long series of unfortunate ISA designs.
  - The dominant ISA in modern computer systems.

You will learn to write MIPS code and implement a MIPS processor. You will learn to read a common subset of x86.
MIPS Basics

• Instructions
  • 4 bytes (32 bits)
  • 4-byte aligned (i.e., they start at addresses that are a multiple of 4 -- 0x0000, 0x0004, etc.)
  • Instructions operate on memory and registers

• Memory Data types (also aligned)
  • Bytes -- 8 bits
  • Half words -- 16 bits
  • Words -- 32 bits
  • Memory is denote “M” (e.g., M[0x10] is the byte at address 0x10)

• Registers
  • 32 4-byte registers in the “register file”
  • Denoted “R” (e.g., R[2] is register 2)

• There’s a handy reference on the inside cover of your text book and a detailed reference in Appendix B.
### Bytes and Words

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA</td>
</tr>
<tr>
<td>0x0001</td>
<td>0x15</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x13</td>
</tr>
<tr>
<td>0x0003</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x0004</td>
<td>0x76</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFFFFE</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFF</td>
<td>.</td>
</tr>
</tbody>
</table>

#### Byte addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA15</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x13FF</td>
</tr>
<tr>
<td>0x0004</td>
<td>.</td>
</tr>
<tr>
<td>0x0006</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFE</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFF</td>
<td>.</td>
</tr>
</tbody>
</table>

#### Half Word Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA15</td>
</tr>
<tr>
<td>0x0002</td>
<td>0x13FF</td>
</tr>
<tr>
<td>0x0004</td>
<td>.</td>
</tr>
<tr>
<td>0x0006</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFE</td>
<td>.</td>
</tr>
<tr>
<td>0xFFFFF</td>
<td>.</td>
</tr>
</tbody>
</table>

#### Word Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0xAA1513FF</td>
</tr>
<tr>
<td>0x0004</td>
<td>.</td>
</tr>
<tr>
<td>0x0008</td>
<td>.</td>
</tr>
<tr>
<td>0x000C</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
</tbody>
</table>

- In modern ISAs (including MIPS) memory is “byte addressable”
- In MIPS, half words and words are aligned.
The MIPS Register File

- All registers are the same
  - Where a register is needed any register will work
- By convention, we use them for particular tasks
  - Argument passing
  - Temporaries, etc.
  - These rules (“the register discipline”) are part of the ISA
- $zero is the “zero register”
  - It is always zero.
  - Writes to it have no effect.

<table>
<thead>
<tr>
<th>Name</th>
<th>number</th>
<th>use</th>
<th>Callee saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>zero</td>
<td>n/a</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>Assemble Temp</td>
<td>no</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2 - 3</td>
<td>return value</td>
<td>no</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4 - 7</td>
<td>arguments</td>
<td>no</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8 - 15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16 - 23</td>
<td>saved temporaries</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24 - 25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$k0 - $k1</td>
<td>26 - 27</td>
<td>Res. for OS</td>
<td>yes</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
MIPS R-Type Arithmetic Instructions

- R-Type instructions encode operations of the form "a = b OP c" where ‘OP’ is +, -, <<, &, etc.
- Bit fields
  - "opcode" encodes the operation type.
  - "funct" specifies the particular operation.
  - "rs" are “rt” source registers; “rd” is the destination register
    - 5 bits can specify one of 32 registers.
- "shamt" is the “shift amount” for shift operations
  - Since registers are 32 bits, 5 bits are sufficient

Examples

- add $t0, $t1, $t2
  - opcode = 0, funct = 0x20
- nor $t0, $t1, $t2
  - opcode = 0, funct = 0x27
- sll $t0, $t1, 4
  - opcode = 0, funct = 0x0, shamt = 4
MIPS R-Type Control Instructions

- R-Type encodes “register-indirect” jumps
- Jump register
  - \texttt{jr rs}: PC = R[rs]
- Jump and link register
  - \texttt{jalr rs, rd}: R[rd] = PC + 8; PC = R[rs]
  - rd default to $ra (i.e., the assembler will fill it in if you leave it out)

### Examples

- \texttt{jr $t2}
  - PC = r[10]
  - opcode = 0, funct = 0x8
- \texttt{jalr $t0}
  - PC = R[8]
  - R[31] = PC + 8
  - opcode = 0, funct = 0x9
- \texttt{jalr $t0, $t1}
  - PC = R[8]
  - R[9] = PC + 8
  - opcode = 0, funct = 0x9
MIPS I-Type Arithmetic Instructions

- I-Type arithmetic instructions encode operations of the form “a = b OP #”
  - ‘OP’ is +, -, <<, &, etc and # is an integer constant
    - More formally, e.g.: R[rd] = R[rs] + 42

- Components
  - “opcode” encodes the operation type.
  - “rs” is the source register
  - “rd” is the destination register

- “immediate” is a 16 bit constant used as an argument for the operation

Examples

- addi $t0, $t1, -42
  - opcode = 0x8

- ori $t0, $zero, 42
  - R[4] = R[0] | 42
  - opcode = 0xd
  - Loads a constant into $t0
MIPS I-Type Branch Instructions

- I-Type also encode branches
  - if (R[rd] OP R[rs])
    \[ PC = PC + 4 + 4 \times \text{Immediate} \]
  - else
    \[ PC = PC + 4 \]

- Components
  - “rs” and “rt” are the two registers to be compared
  - “rt” is sometimes used to specify branch type.

- “immediate” is a 16 bit branch offset
  - It is the signed offset to the target of the branch
  - Limits branch distance to 32K instructions
  - Usually specified as a label, and the assembler fills it in for you.

Examples

- \text{beq} $t0$, $t1$, $-42$
    \[ PC = PC + 4 + 4 \times -42 \]
  - \text{opcode} = 0x4

- \text{bgez} $t0$, $-42$
  - if \( R[8] \geq 0 \)
    \[ PC = PC + 4 + 4 \times -42 \]
  - \text{opcode} = 0x1
  - rt = 1
MIPS I-Type Memory Instructions

- I-Type also encode memory access
  - Store: $M[R[rs] + \text{Immediate}] = R[rt]$
  - Load: $R[rt] = M[R[rs] + \text{Immediate}]$
- MIPS has load/stores for byte, half word, and word
- Sub-word loads can also be signed or unsigned
  - Signed loads sign-extend the value to fill a 32 bit register.
  - Unsigned zero-extend the value.
- “immediate” is a 16 bit offset
  - Useful for accessing structure components
  - It is signed.

Examples

- **lw $t0, 4($t1)**
  - opcode = 0x23
- **sb $t0, -17($t1)**
  - opcode = 0x28
MIPS J-Type Instructions

- J-Type encodes the jump instructions
- Plain Jump
  - $\text{JumpAddress} = \{\text{PC}+4[31:28],\text{Address},2'b0\}$
  - Address replaces *most* of the PC
  - $\text{PC} = \text{JumpAddress}$
- Jump and Link
  - $R[ra] = \text{PC} + 8; \text{PC} = \text{JumpAddress}$
- J-Type also encodes misc instructions
  - syscall, interrupt return, and break (more later)

Examples

- $j \ $t0
  - $\text{PC} = R[8]$
  - $\text{opcode} = 0x2$
- $jal \ $t0
  - $R[31] = \text{PC} + 8$
  - $\text{PC} = R[8]$
Executing a MIPS program

- All instructions have
  - $\leq 1$ arithmetic op
  - $\leq 1$ memory access
  - $\leq 2$ register reads
  - $\leq 1$ register write
  - $\leq 1$ branch

- All instructions go through all the steps

- As a result
  - Implementing MIPS is (sort of) easy!
  - The resulting HW is (relatively) simple!
MIPS Mystery 1: Delayed Loads

• The value retrieved by a load is not available to the next instruction.

Example

ori $t0, $zero, 4
sw $t0, 0($sp)
lw $t1, 0($sp)
or $t2, $t1, $zero
or $t3, $t1, $zero

$t2 == 0
$t3 == 4

file: delayed_load.s
MIPS Mystery 1: Delayed Loads

- The value retrieved by a load is not available to the next instruction.

Example

```assembly
ori $t0, $zero, 4
sw $t0, 0($sp)
lw $t1, 0($sp)
or $t2, $t1, $zero
or $t3, $t1, $zero
```

$t2 == 0
$t3 == 4

file: delayed_load.s

Why? We’ll talk about it in a few weeks.
MIPS Mystery 2: Delayed Branches

• The instruction after the branch executes even if the branch is taken.
• All jumps and branches are delayed -- the next instruction *always* executes.

Example

```
ori $t0, $zero, 4
beq $t0, $t0, foo
ori $t0, $zero, 5
foo:
    $t0 == 5
```

file: delayed_branch.s
MIPS Mystery 2: Delayed Branches

- The instruction after the branch executes even if the branch is taken.
- All jumps and branches are delayed -- the next instruction always executes

Example

```
ori $t0, $zero, 4
beq $t0, $t0, foo
ori $t0, $zero, 5
foo:
    $t0 == 5
```

Why? We’ll talk about it in a few weeks.
Live Demo!

Source code available on the course web site
Example 1: add.s

<table>
<thead>
<tr>
<th>inst</th>
<th>bits</th>
<th>inst</th>
<th>source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>[00400000] 01444820</td>
<td>add $9, $10, $4</td>
<td>; 2: add $t1, $t2, $a0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>5 bits</th>
<th>6 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x9</td>
<td>0xa</td>
<td>0x4</td>
<td>0</td>
<td>0x20</td>
</tr>
</tbody>
</table>

=

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>01010</td>
<td>00100</td>
<td>01001</td>
<td>00000</td>
<td>100000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

31 26 25 21 20 16 15 11 10 6 5 0
Example: Warts

• Files
  • delayed_branch.s
  • delayed_load.s

• Make sure to set SPIM settings to “bare machine”
  • See the SPIM tutorial
  • Always check that you’ve got this set. We will not be using “simple machine” in this class.
Example: conditional.s

\[
i = 42
\]

if \( i \& 7 \)
\[
i += 8
\]
else
\[
i += 4
\]

\[
\text{ori } $t0, $zero, 42
\]
\[
\text{andi } $t1, $t0, 7
\]
\[
\text{beq } $t1, $zero, ifcode
\]
\[
\text{add } $zero, $zero, $zero
\]

elsecode:
\[
\text{addi } $t0, $t0, 4
\]
\[
\text{beq } $zero, $zero, followon
\]
\[
\text{add } $zero, $zero, $zero
\]

ifcode:
\[
\text{addi } $t0, $t0, 8
\]

followon:

\[
$t0 \text{ is } i
\]
Example: loop.s

do
    j += i
while i != 0

$t0$ is $i$
$t1$ is $j$

[00400000] 34080005 ori $8$, $0$, 5 ; 1: ori $t0$, $zero$, 5
[00400004] 01284820 add $9$, $9$, $8$ ; 3: add $t1$, $t1$, $t0$
[00400008] 2108ffff addi $8$, $8$, $-1$ ; 4: addi $t0$, $t0$, $-1$
[0040000c] 1500fffe bne $8$, $0$, $-8$ [top-0x0040000c]; 5: bne $t0$, $zero$, top
[00400010] 00000020 add $0$, $0$, $0$ ; 6: add $zer$, $zero$, $zero$ #noop in the branch delay slot.
Function Calls

• Challenges
  • Passing in i and calling lg
  • Returning the sum
  • Continuing execution after the call
  • Allocating temporaries
  • Releasing temporaries

Example

```c
int lg(int i) {
  if (i)
    return lg(i >> 1) + 1;
  else
    return 0;
}
```
Calling and Returning

- Passing arguments
  - The first 4 in $a0...$a3
  - Any more go on the stack
- Invoking the function
  - jal <label>
  - Stores PC + 8 in $ra
- Return value in $v0
- Return to caller
  - jr $ra

Example

```assembly
ori $a0, $zero, 4  
jal log2
addi $zero, $zero, 0
... access $v0 ...
log2:
...
ori $v0, $zero, 0
jr $ra
```
Managing Registers

- Sharing registers
  - A called function will modify registers
  - The caller needs to keep some values around.
- The ISA specifies which registers a function can modify
- A function can use “callee-saved” registers, but must restore their value.

<table>
<thead>
<tr>
<th>Name</th>
<th>number</th>
<th>use</th>
<th>Callee saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>zero</td>
<td>n/a</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>Assemble Temp</td>
<td>no</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2 - 3</td>
<td>return value</td>
<td>no</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4 - 7</td>
<td>arguments</td>
<td>no</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8 - 15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16 - 23</td>
<td>saved temporaries</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24 - 25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$k0 - $k1</td>
<td>26 - 27</td>
<td>Res. for OS</td>
<td>yes</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame ptr</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
The Stack

• The stack provides local storage for function calls (e.g., for preserving registers)
  • Local variables
  • Register overflow
  • Preserved register contents

• It is as first-in-last-out (FILO) queue
• For historical the stack grows down from high memory addresses to low.
• The stack pointer ($sp) points to the “top” of the stack.
Preserving Registers

Assume $ra = 0xBEEF

To save $ra:
   addi $sp, $sp, -4
   sw $ra, 0($sp)

... function calls ...

To restore $ra:
   lw $ra, 0($sp)
   addi $sp, $sp, 4

High Memory

Low Memory

$sp →

???
Preserving Registers

Assume $ra = 0xBEEF

To save $ra:
- addi $sp, $sp, -4
- sw $ra, 0($sp)
  ... function calls ...
To restore $ra:
- lw $ra, 0($sp)
- addi $sp, $sp, 4

$sp

High Memory

Low Memory
Preserving Registers

Assume $ra = 0xBEEF

To save $ra:
  addi $sp, $sp, -4
  sw $ra, 0($sp)

... function calls ...

To restore $ra:
  lw $ra, 0($sp)
  addi $sp, $sp, 4
Preserving Registers

Assume $ra = 0xBEEF

To save $ra:
   addi $sp, $sp, -4
   sw $ra, 0($sp)
   ... function calls ...

To restore $ra:
   lw $ra, 0($sp)
   addi $sp, $sp, 4
Preserving Registers

Assume $ra = 0xBEEF

To save $ra:
  addi $sp, $sp, -4
  sw $ra, 0($sp)

... function calls ...

To restore $ra:
  lw $ra, 0($sp)
  addi $sp, $sp, 4

<table>
<thead>
<tr>
<th>High Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>???</td>
</tr>
<tr>
<td>0xBEEF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Low Memory</th>
</tr>
</thead>
</table>
Preserving Registers

Assume $ra = 0xBEEF

To save $ra:
   addi $sp, $sp, -4
   sw $ra, 0($sp)

... function calls ...

To restore $ra:
   lw $ra, 0($sp)
   addi $sp, $sp, 4

Note that $sp is also restored
int lg(int i) {
    // Save registers
    if (i)
        return lg(i >> 1) + 1;
    else
        return 0;
    // Restore registers
}

lg:
    addi $sp, $sp, -4
    sw $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
```
int lg(int i) {
    // Save registers
    if (i)
        return lg(i >> 1) + 1;
    else
        return 0;
    // Restore registers
}
```
lg:
    addi $sp, $sp, -4
    sw $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero

int lg(int i) {
    // Save registers
    if (i)
        return lg(i >> 1) + 1;
    else
        return 0;
    // Restore registers
}
int lg(int i) {
    // Save registers
    if (i)
        return lg(i >> 1) + 1;
    else
        return 0;
    // Restore registers
}
int lg(int i) {
    // Save registers
    if (i)
        return lg(i >> 1) + 1;
    else
        return 0;
    // Restore registers
}

lg:
    addi $sp, $sp, -4
    sw $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
int lg(int i) {
    // Save registers
    if (i)
        return lg(i >> 1) + 1;
    else
        return 0;
    // Restore registers
}

lg:
    addi $sp, $sp, -4
    sw $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
int lg(int i) {
    // Save registers
    if (i)
        return lg(i >> 1) + 1;
    else
        return 0;
    // Restore registers
}

lg:
    addi $sp, $sp, -4
    sw $ra, 0($sp)

    bne $a0, $zero, big
    add $zero, $zero, $zero
    ori $v0, $zero, 0
    j end
    add $zero, $zero, $zero

big:
    srl $a0, $a0, 1
    jal lg
    add $zero, $zero, $zero
    addi $v0, $v0, 1

end:
    lw $ra, 0($sp)
    addi $sp, $sp, 4

    jr $ra
    add $zero, $zero, $zero
int lg(int i) {
    // Save registers
    if (i)
        return lg(i >> 1) + 1;
    else
        return 0;
    // Restore registers
}

lg:
    addi $sp, $sp, -4
    sw $ra, 0($sp)

bne $a0, $zero, big
add $zero, $zero, $zero
ori $v0, $zero, 0
j end
add $zero, $zero, $zero

big:
srl $a0, $a0, 1
jal lg
add $zero, $zero, $zero
addi $v0, $v0, 1

end:
lw $ra, 0($sp)
addi $sp, $sp, 4

jr $ra
add $zero, $zero, $zero
```c
int lg(int i) {
    // Save registers
    if (i)
        return lg(i >> 1) + 1;
    else
        return 0;
    // Restore registers
}
```
int lg(int i) {
    // Save registers
    if (i)
        return lg(i >> 1) + 1;
    else
        return 0;
    // Restore registers
}
int lg(int i) {
    // Save registers
    if (i) return lg(i >> 1) + 1;
    else return 0;
    // Restore registers
}

Delay slots
Live Demo!

Source code available on the class web site

Slides/01 ISA Part-I examples/release/lg.s
Slides/01 ISA Part-I examples/release/lg.c
Slides/01 ISA Part-I examples/release/lg-opt.s
Filling Delay Slots

• Compilers put useful instructions in delay slots.
• Branch delay
  • Use instructions from before the branch.
• Load delay
  • Use an instruction that doesn’t need the loaded value
  • Or that needs the old value of the register

lg:
  addi $sp, $sp, -4
  bne $a0, $zero, big
  sw $ra, 0($sp)
  j end
  ori $v0, $zero, 0
big:
  jal lg
  srl $a0, $a0, 1
  addi $v0, $v0, 1
end:
  lw $ra, 0($sp)
  addi $sp, $sp, 4
  jr $ra
  add $zero, $zero, $zero
Filling Delay Slots

- Compilers put useful instructions in delay slots.
- Branch delay
  - Use instructions from before the branch.
- Load delay
  - Use an instruction that doesn’t need the loaded value
  - Or that needs the old value of the register

```
lg:
  addi $sp, $sp, -4
  bne $a0, $zero, big
  sw $ra, 0($sp)
  j end
  ori $v0, $zero, 0
big:
  jal lg
  srl $a0, $a0, 1
  addi $v0, $v0, 1
end:
  lw $ra, 0($sp)
  addi $sp, $sp, 4
  jr $ra
  add $zero, $zero, $zero
```
Filling Delay Slots

- Compilers put useful instructions in delay slots.
- **Branch delay**
  - Use instructions from before the branch.
- **Load delay**
  - Use an instruction that doesn’t need the loaded value
  - Or that needs the old value of the register

```
lg:
  addi $sp, $sp, -4
  bne $a0, $zero, big
  sw $ra, 0($sp)
  j end
  ori $v0, $zero, 0
big:
  jal lg
  srl $a0, $a0, 1
  addi $v0, $v0, 1
end:
  lw $ra, 0($sp)
  addi $sp, $sp, 4
  jr $ra
  add $zero, $zero, $zero
```
Pseudo Instructions

- Assembly language programming is repetitive
- Some code is not very readable
- The assembler provides some simple shorthand for common operations
- Register $at is reserved for implementing them.

<table>
<thead>
<tr>
<th>Assembly</th>
<th>Shorthand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>or $s1, $zero, $s2</td>
<td>mov $s1, $s2</td>
<td>move</td>
</tr>
<tr>
<td>beq $zero, $zero, &lt;label&gt;</td>
<td>b &lt;label&gt;</td>
<td>unconditional branch</td>
</tr>
<tr>
<td>Homework?</td>
<td>li $s2, &lt;value&gt;</td>
<td>load 32 bit constant</td>
</tr>
<tr>
<td>Homework?</td>
<td>nop</td>
<td>do nothing</td>
</tr>
<tr>
<td>Homework?</td>
<td>div d, s1, s2</td>
<td>dst = src1/src2</td>
</tr>
<tr>
<td>Homework?</td>
<td>mulou d, s1, s2</td>
<td>dst = low32bits(src1*src2)</td>
</tr>
</tbody>
</table>
Declaring Variables

• Assembler directives declare static variables
  • The reside in the “.data” section
  • Code is in the “.text” section
• Labels allow access
  • Use la (load address)
• More details in B.10 in the text

Example

```assembly
.data
a_str:
    .ascii "Hello!"
str_len:
    .word 6
    .align 2
some_letter:
    .byte 'l'
.text
main:
    la $a0, a_str
    ...access via $a0...
example: count.s```

```assembly
```

37
### Labels in the Assembler

<table>
<thead>
<tr>
<th>Address</th>
<th>Bytes</th>
<th>Raw Insts.</th>
<th>Asm. Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>count:</td>
<td></td>
<td><code>lui $1, 4097 [some_letter]</code></td>
<td><code>lui $1, 4097 [some_letter]</code></td>
</tr>
<tr>
<td>la $t0, foo</td>
<td></td>
<td><code>ori $4, $1, 12 [some_letter]</code></td>
<td><code>la $a0, some_letter</code></td>
</tr>
<tr>
<td>la $t1, some_letter</td>
<td></td>
<td><code>lbu $12, 0($t1)</code></td>
<td><code>lbu $t4, 0($t1)</code></td>
</tr>
<tr>
<td>lbu $a0, 0($t1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>la $t2, str_len</td>
<td></td>
<td><code>lui $1, 4097 [str_len]</code></td>
<td><code>lui $1, 4097 [str_len]</code></td>
</tr>
<tr>
<td>lbu $a1, 0($t2)</td>
<td></td>
<td><code>ori $5, $1, 8 [str_len]</code></td>
<td><code>la $a1, str_len</code></td>
</tr>
<tr>
<td>beq $a0, $zero, count</td>
<td></td>
<td><code>lbu $13, 0($13)</code></td>
<td><code>lbu $t5, 0($t5)</code></td>
</tr>
<tr>
<td>add $zero, $zero, $zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne $a1, $zero, done</td>
<td></td>
<td><code>addi $9, $9, 1</code></td>
<td><code>done:</code></td>
</tr>
<tr>
<td>add $zero, $zero, $zero</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $t1, $t1, 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>done:</td>
<td></td>
<td><code>jal 0x00400000 [count]</code></td>
<td><code>jal count</code></td>
</tr>
<tr>
<td>jal count</td>
<td></td>
<td><code>addi $9, $9, 1</code></td>
<td></td>
</tr>
<tr>
<td>add $zero, $zero, $zero</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Data Section

- **foo:**
  - `.asci "Hello!"
- **str_len:**
  - `.word 7`
- **some_letter:**
  - `.byte 'l'`

### Address

<table>
<thead>
<tr>
<th></th>
<th>Bytes</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo:</td>
<td>0x10010000</td>
<td>&quot;Hello!&quot;</td>
</tr>
<tr>
<td>str_len:</td>
<td>0x10010008</td>
<td></td>
</tr>
<tr>
<td>some_letter:</td>
<td>0x1001000c</td>
<td></td>
</tr>
</tbody>
</table>

### Calculations

- `foo: 0x10010000 = (4097 << 16) | 0`
- `str_len: 0x10010008 = (4097 << 16) | 8`
- `some_letter: 0x1001000c = (4097 << 16) | 12`
From C to MIPS
Compiling: C to bits

Architecture-independent

Your Brain

Brain/Fingers/SWE

Programming Languages (C, C++)

Compiler

Architecture-dependent

Assembly Language

Assembler

Machine code (.o files)

Linker

Executable (.exe files)
Count the number of 1’s in the binary representation of i

```c
int popcount(int i) {
    int c = 0;
    int j;
    for(j = 0; j < 32; j++) {
        if (i & (1 << j))
            c++;
    }
    return c;
}
```
In the Compiler

```
int popcount(int i) {
    int c = 0;
    int j;
    for(j = 0; j < 32; j++) {
        if (i & (1 << j)) {
            c++;
        }
    }
    return c;
}
```
Function popcount

Arguments
int i
int c
int j

Body
= =
for
return c

= = <
= = +
if
= &
= = +
&
= = <<

Abstract Syntax Tree

Control Flow Graph
In the Compiler

```assembly
popcount:
    ori $v0, $zero, 0
    ori $t1, $zero, 0

top:
    slti $t2, $t1, 32
    beq $t2, $zero, end
    nop
    addi $t3, $zero, 1
    sllv $t3, $t3, $t1
    and $t3, $a0, $t3
    beq $t3, $zero, notone
    nop
    addi $v0, $v0, 1

notone:
    beq $zero, $zero, top
    addi $t1, $t1, 1

end:
    jr $ra
    nop
```

Control flow graph
In the Assembler

**popcount:**

- ori $v0, $zero, 0
- ori $t1, $zero, 0

**top:**

- slti $t2, $t1, 32
- beq $t2, $zero, end
- nop
- addi $t3, $zero, 1
- sllv $t3, $t3, $t1
- and $t3, $a0, $t3
- beq $t3, $zero, notone
- nop
- addi $v0, $v0, 1

**notone:**

- beq $zero, $zero, top
- addi $t1, $t1, 1

**end:**

- jr $ra
- nop

---

Assembly

Executable Binary
```c
int popcount(int i) {
    int c = 0;
    int j;
    for(j = 0; j < 32; j++) {
        if (i & (1 << j))
            c++;
    }
    return c;
}
```

```
int popcount(int i) {
    ori $v0, $zero, 0
    ori $t1, $zero, 0

    top:
        slti $t2, $t1, 32
        beq  $t2, $zero, end
        nop
        addi $t3, $zero, 1
        sllv $t3, $t3, $t1
        and $t3, $a0, $t3
        beq $t3, $zero, notone
        nop
        addi $v0, $v0, 1
        notone:
            beq $zero, $zero, top
            addi $t1, $t1, 1
        end:
            jr $ra
            nop
```
Top 5 Reasons to Use Assembly

1. You are writing a compiler, so you have no choice.
2. You want to understand what the machine is actually doing (e.g., why your code is slow). In this case, you just need to read assembly.
3. You need to do things that are not possible in C
   - e.g., It is not possible to implement locks correctly in C.
   - e.g., Many other low-level OS operations can’t be expressed in C.
4. It’s faster sometimes
   - Compilers mechanically convert C to assembly, and they may not emit the fastest code possible.
   - You might know better...
     - The compiler might not recognize opportunities to apply specialized instructions (e.g., SSE vector instructions)
     - You might be desperate for performance, and be able to squeeze a bit out here or there.
   - But probably not.
     - Modern compilers are very good.
     - Unless you know exactly why you want to use assembly, you shouldn’t.
     - Even then, you should try to find a way to do it in C (e.g., Compiler “intrinsics” to force the compiler to emit SSE instructions, or restructuring your C code)
5. You are doing cse141 homework