Great Ideas in Computer Architecture

Introduction to Machine Language

Instructor: Steven Ho
Review of Last Lecture

• C Memory Layout
  – Local variables disappear because Stack changes
  – Global variables don’t disappear because they are in Static Data
  – Dynamic memory available using `malloc` and `free`, but must be used VERY CAREFULLY
EVERY COMMAND IS DIRECTLY IMPLEMENTED IN THE HARDWARE

```
\text{temp} = \nu[k];
\nu[k] = \nu[k+1];
\nu[k+1] = \text{temp};
```

**Great Idea #1: Levels of Representation/Interpretation**

![Diagram showing the process from higher-level language to machine language](image)

- **Higher-Level Language Program (e.g. C)**
  - **Compiler**
  - **Assembly Language Program (e.g. MIPS)**
  - **Assembler**
  - **Machine Language Program (MIPS)**
- **Machine Interpretation**
- **Hardware Architecture Description (e.g. block diagrams)**
- **Architecture Implementation**
- **Logic Circuit Description (Circuit Schematic Diagrams)**

**We were here**

**We are now here**
# Mainstream ISAs

## X86
- **Designer**: Intel, AMD
- **Bits**: 16-bit, 32-bit and 64-bit
- **Introduced**: 1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
- **Design**: CISC
- **Type**: Register-memory
- **Encoding**: Variable (1 to 15 bytes)
- **Endianness**: Little

## ARM Architectures
- **Designer**: ARM Holdings
- **Bits**: 32-bit, 64-bit
- **Introduced**: 1985; 31 years ago
- **Design**: RISC
- **Type**: Register-Register
- **Encoding**: AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility[1]
- **Endianness**: Big (little as default)

## RISC-V
- **Designer**: University of California, Berkeley
- **Bits**: 32, 64, 128
- **Introduced**: 2010
- **Version**: 2.2
- **Design**: RISC
- **Type**: Load-store
- **Encoding**: Variable
- **Branching**: Compare-and-branch
- **Endianness**: Little

---

Macbooks & PCs (Core i3, i5, i7, M)
x86 Instruction Set

Smartphone-like devices (iPhone, iPad, Raspberry Pi)
[ARM Instruction Set](#)

Versatile and open-source
Relatively new, designed for cloud computing, high-end phones, small embedded sys.
[RISCV Instruction Set](#)
Complex/Reduced Instruction Set Computing

• Early trend: add more and more instructions to do elaborate operations – *Complex Instruction Set Computing* (CISC)
  – difficult to learn and comprehend language
  – super-complicated (slow?) hardware

• Opposite philosophy later began to dominate: *Reduced Instruction Set Computing* (RISC)
  – Simpler (and smaller) instruction set makes it easier to build fast hardware
  – Let software do the complicated operations by composing simpler ones
# Mainstream ISAs

**Intel x86**
- **Designer**: Intel, AMD
- **Bits**: 16-bit, 32-bit and 64-bit
- **Introduced**: 1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
- **Design**: CISC
- **Type**: Register-memory
- **Encoding**: Variable (1 to 15 bytes)
- **Endianness**: Little

**ARM Architectures**
- **Designer**: ARM Holdings
- **Bits**: 32-bit, 64-bit
- **Introduced**: 1985; 31 years ago
- **Design**: RISC
- **Type**: Register-Register
- **Encoding**: AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) uses mixed 16- and 32-bit instructions. ARMv7 user-space compatibility
- **Endianness**: Big (little as default)

**RISC-V**
- **Designer**: University of California, Berkeley
- **Bits**: 32, 64, 128
- **Introduced**: 2010
- **Version**: 2.2
- **Design**: RISC
- **Type**: Load-store
- **Encoding**: Variable
- **Branching**: Compare-and-branch
- **Endianness**: Little

---

- **Macbooks & PCs** (Core i3, i5, i7, M)
  - x86 Instruction Set

- **Smartphone-like devices** (iPhone, iPad, Raspberry Pi)
  - ARM Instruction Set

- **Versatile and open-source**
  - Relatively new, designed for cloud computing, high-end phones, small embedded sys.
  - RISCV Instruction Set
RISC-V!

The RISC-V Instruction Set Architecture

RISC-V (pronounced "risk-five") is a new instruction set architecture (ISA) that was originally designed to support computer architecture research and education, which we now hope will become a standard open architecture for industry implementations. RISC-V was originally developed in the Computer Science Division of the EECS Department at the University of California, Berkeley.
RISC-V Manual

http://inst.eecs.berkeley.edu/~cs61c/su18/img/riscvcard.pdf (Instruction Set)

http://digitalassets.lib.berkeley.edu/techreports/ucb/text/EECS-2016-1.pdf (Architecture)
Variables in Hardware

• Already know instructions must be directly implemented in hardware
• What about variables?!?!
  – Live in Memory
  – Relative to your processor, memory is sooooo slooooooowwwwwwwwwwwwwww
  – *predetermined* small number of fast *registers* in processor to hold variables.
Registers vs. Memory

• What if more variables than registers?
  – Keep most frequently used in registers and move the rest to memory (called spilling to memory)

• Why not all variables in memory?
  – Smaller is faster: registers 100-500 times faster
  – Registers more versatile
    • In 1 arithmetic instruction: read 2 operands, perform 1 operation, and 1 write
    • In 1 data transfer instruction: 1 memory read/write, no operation
RISCV -- How Many Registers?

• Tradeoff between speed and availability
  – more registers → can house more variables simultaneously; all registers are slower.
• RISCV has 32 registers (x0-x31)
  – Each register is 32 bits wide and holds a word
RISCV Registers

• Register denoted by ‘x’ can be referenced by number (x0-x31) or name:
  – Registers that hold programmer variables:
    s0-s1 ↔ x8-x9
    s2-s11 ↔ x18-x27
  – Registers that hold temporary variables:
    t0-t2 ↔ x5-x7
    t3-t6 ↔ x28-x31
  – You’ll learn about the other 13 registers later

• Registers have no type (C concept); the operation being performed determines how register contents are treated
Registers -- Summary

• In high-level languages, number of variables limited only by available memory
• ISAs have a fixed, small number of operands called registers
  – Special locations built directly into hardware
  – **Benefit:** Registers are EXTREMELY FAST (faster than 1 billionth of a second)
  – **Drawback:** Operations can only be performed on these predetermined number of registers
Registers in a Computer

- We begin our study of how a computer works!
  - Control
  - Datapath
  - Memory
  - Input
  - Output

- Registers are part of the Datapath
Adminstrivia

- HW0 & Minibios due TONIGHT at midnight
- HW1 (C) due Thursday (6/28)
- HW2 (RISCV) released tonight, due Monday, 7/2
- Proj1 (Chatroom) due Friday (6/29)
- Project Party TONIGHT (the Woz lounge in 430 Soda; 4PM - 6PM)
- MT1 Review this Saturday (6/30) Soda 306 2-4P
- Guerilla Session, Sunday (7/1) Cory 540AB 2-4P
- No lecture or labs on July 4th
- Tutoring sessions start today!
- bring iClickers for attendance starting tomorrow!
Please read the labs! :)
RISCV Agenda

• Basic Arithmetic Instructions
• Comments
• x0 (zero)
• Immediates
• Data Transfer Instructions
• Decision Making Instructions
• Bonus: C to RISCV Practice
• Bonus: Additional Instructions
RISCV Instructions (1/2)

• Instruction Syntax is rigid:

\[
\text{op} \ \text{dst}, \ \text{src1}, \ \text{src2}
\]

– 1 operator, 3 operands
  • \text{op} = \text{operation name ("operator")}
  • \text{dst} = \text{register getting result ("destination")}
  • \text{src1} = \text{first register for operation ("source 1")}
  • \text{src2} = \text{second register for operation ("source 2")}

• Keep hardware simple via regularity
RISCV Instructions (2/2)

• One operation per instruction, at most one instruction per line

• Assembly instructions are related to C operations (\(=, +, -, *, \), \(/, \), \&, \|, etc.)
  – Must be, since C code decomposes into assembly!
  – A single line of C may break up into several lines of RISCV
RISCV Instructions Example

• Your very first instructions!
  (assume here that the variables \(a\), \(b\), and \(c\) are assigned to registers \(s1\), \(s2\), and \(s3\), respectively)

• Integer Addition (\(\text{add}\))
  – C: \(a = b + c\)
  – RISCV: \(\text{add } s1, s2, s3\)

• Integer Subtraction (\(\text{sub}\))
  – C: \(a = b - c\)
  – RISCV: \(\text{sub } s1, s2, s3\)
RISCV Instructions Example

• Suppose $a \rightarrow s0, b \rightarrow s1, c \rightarrow s2, d \rightarrow s3$ and $e \rightarrow s4$. Convert the following C statement to RISCV:

$$a = (b + c) - (d + e);$$

add $t1, s3, s4$

add $t2, s1, s2$

sub $s0, t2, t1$

Ordering of instructions matters (must follow order of operations)

Utilize temporary registers
RISCV Agenda

• Basic Arithmetic Instructions
• Comments
• x0 (zero)
• Immediates
• Data Transfer Instructions
• Decision Making Instructions
• Bonus: C to RISCV Practice
• Bonus: Additional Instructions
Comments in RISCV

- Comments in RISCV follow hash mark (#) until the end of line
  - Improves readability and helps you keep track of variables/registers!

```
add t1, s3, s4  # t1=d+e
add t2, s1, s2  # t2=b+c
sub s0, t2, t1  # a=(b+c)-(d+e)
```
RISCV Agenda

- Basic Arithmetic Instructions
- Comments
- x0 (zero)
- Immediates
- Data Transfer Instructions
- Decision Making Instructions
- Bonus: C to RISCV Practice
- Bonus: Additional Instructions
The Zero Register

• Zero appears so often in code and is so useful that it has its own register!

• Register zero (x0 or zero) always has the value 0 and cannot be changed!
  – i.e. any instruction with x0 as dst has no effect

• Example Uses:
  – add s3, x0, x0  # c=0
  – add s1, s2, x0  # a=b
RISCV Agenda

• Basic Arithmetic Instructions
• Comments
• x0 (zero)
  • **Immediates**
• Data Transfer Instructions
• Decision Making Instructions
• Bonus: C to RISCV Practice
• Bonus: Additional Instructions
**Immediates**

- Numerical constants are called **immediates**
- Separate instruction syntax for immediates:
  
  \[
  \text{opi dst, src, imm}
  \]
  
  - Operation names end with ‘i’, replace 2\textsuperscript{nd} source register with an immediate

- **Example Uses:**
  
  - \text{addi s1, s2, 5}  \# a=b+5
  - \text{addi s3, s3, 1}  \# c++

- **Why no subi instruction?**
### RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

<table>
<thead>
<tr>
<th>MNEMONIC</th>
<th>FMT</th>
<th>NAME</th>
<th>DESCRIPTION (in Verilog)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, addw</td>
<td>R</td>
<td>ADD (Word)</td>
<td>R[rd] = R[rs1] + R[rs2]</td>
</tr>
<tr>
<td>addi, addiw</td>
<td>I</td>
<td>ADD Immediate (Word)</td>
<td>R[rd] = R[rs1] + imm</td>
</tr>
<tr>
<td>and</td>
<td>R</td>
<td>AND</td>
<td>R[rd] = R[rs1] &amp; R[rs2]</td>
</tr>
<tr>
<td>andi</td>
<td>I</td>
<td>AND Immediate</td>
<td>R[rd] = R[rs1] &amp; imm</td>
</tr>
<tr>
<td>auipc</td>
<td>U</td>
<td>Add Upper Immediate to PC</td>
<td>R[rd] = PC + imm(imm, 12'b0)</td>
</tr>
<tr>
<td>beq</td>
<td>SB</td>
<td>Branch Equal</td>
<td>if(R[rs1] == R[rs2]) PC = PC + {imm, 1'b0}</td>
</tr>
<tr>
<td>bge</td>
<td>SB</td>
<td>Branch Greater than or Equal</td>
<td>if(R[rs1] &gt;= R[rs2]) PC = PC + {imm, 1'b0}</td>
</tr>
<tr>
<td>bgeu</td>
<td>SB</td>
<td>Branch ≥ Unsigned</td>
<td>if(R[rs1] &gt;= R[rs2]) PC = PC + {imm, 1'b0}</td>
</tr>
<tr>
<td>blt</td>
<td>SB</td>
<td>Branch Less Than</td>
<td>if(R[rs1] &lt; R[rs2]) PC = PC + {imm, 1'b0}</td>
</tr>
<tr>
<td>bltu</td>
<td>SB</td>
<td>Branch Less Than Unsigned</td>
<td>if(R[rs1] &lt; R[rs2]) PC = PC + {imm, 1'b0}</td>
</tr>
<tr>
<td>bne</td>
<td>SB</td>
<td>Branch Not Equal</td>
<td>if(R[rs1] != R[rs2]) PC = PC + {imm, 1'b0}</td>
</tr>
<tr>
<td>csrrc</td>
<td>I</td>
<td>Cont./Stat.RegRead&amp;Clear</td>
<td>R[rd] = CSR; CSR = CSR &amp; ~R[rs1]</td>
</tr>
<tr>
<td>csrrs</td>
<td>I</td>
<td>Cont./Stat.RegRead&amp;Set</td>
<td>R[rd] = CSR; CSR = CSR</td>
</tr>
<tr>
<td>csrrsi</td>
<td>I</td>
<td>Cont./Stat.RegRead&amp;Set Immm</td>
<td>R[rd] = CSR; CSR = CSR</td>
</tr>
<tr>
<td>csrrw</td>
<td>I</td>
<td>Cont./Stat.RegRead&amp;Write</td>
<td>R[rd] = CSR; CSR = R[rs1]</td>
</tr>
<tr>
<td>csrrwi</td>
<td>I</td>
<td>Cont./Stat.RegRead&amp;Write Immm</td>
<td>R[rd] = CSR; CSR = imm(imm)</td>
</tr>
<tr>
<td>ebreak</td>
<td>I</td>
<td>Environment BREAK</td>
<td>Transfer control to debugger</td>
</tr>
<tr>
<td>escall</td>
<td>I</td>
<td>Environment CALL</td>
<td>Transfer control to operating system</td>
</tr>
<tr>
<td>fence</td>
<td>I</td>
<td>Synch thread</td>
<td>Synchronizes threads</td>
</tr>
<tr>
<td>fence.i</td>
<td>I</td>
<td>Synch Instr &amp; Data</td>
<td>Synchronizes writes to instruction stream</td>
</tr>
<tr>
<td>jal</td>
<td>UJ</td>
<td>Jump &amp; Link</td>
<td>R[rd] = PC+4; PC = PC + imm(imm, 1'b0)</td>
</tr>
<tr>
<td>jalr</td>
<td>I</td>
<td>Jump &amp; Link Register</td>
<td>R[rd] = PC+4; PC = R[rs1] + imm</td>
</tr>
<tr>
<td>ld</td>
<td>I</td>
<td>Load Doubleword</td>
<td>R[rd] = M[R[rs1]] + imm(63:0)</td>
</tr>
<tr>
<td>lh</td>
<td>I</td>
<td>Load Halfword</td>
<td>R[rd] = M[R[rs1]] + imm(15:0)</td>
</tr>
<tr>
<td>lhu</td>
<td>I</td>
<td>Load Halfword Unsigned</td>
<td>R[rd] = M[R[rs1]] + imm(15:0)</td>
</tr>
<tr>
<td>lui</td>
<td>U</td>
<td>Load Upper Immediate</td>
<td>R[rd] = {32'bimm31-&gt;, imm, 12'b0}</td>
</tr>
<tr>
<td>lw</td>
<td>I</td>
<td>Load Word</td>
<td>R[rd] = M[R[rs1]] + imm(31:0)</td>
</tr>
</tbody>
</table>
RISCV Agenda

• Basic Arithmetic Instructions
• Comments
• $zero
• Immediates / Overflow
• Data Transfer Instructions
• Decision Making Instructions
• Bonus: C to RISCV Practice
• Bonus: Additional Instructions
Five Components of a Computer

- Data Transfer instructions are between registers (Datapath) and Memory
  - Allow us to fetch and store operands in memory
Data Transfer

• C variables map onto registers; What about large data structures like arrays?
  – Don’t forget memory, our one-dimensional array indexed by addresses starting at 0

• RISCV instructions only operate on registers!

• Specialized data transfer instructions move data between registers and memory
  – Store: register TO memory
  – Load: register FROM memory
Data Transfer

• Instruction syntax for data transfer:

  \[
  \text{memop reg, off(bAddr)}
  \]

  – \text{memop} = \text{operation name ("operator")}
  – \text{reg} = \text{register for operation source or destination}
  – \text{bAddr} = \text{register with pointer to memory ("base address")}
  – \text{off} = \text{address offset (immediate) in bytes ("offset")}

• Accesses memory at address \text{bAddr+off}

• \textbf{Reminder:} A register holds a word of raw data (no type) – make sure to use a register (and offset) that point to a valid memory address
Memory is Byte-Addressed

• What was the smallest data type we saw in C?
  – A char, which was a byte (8 bits)
  – Everything in multiples of 8 bits (e.g. 1 word = 4 bytes)

• Memory addresses are indexed by bytes, not words

• Word addresses are 4 bytes apart
  – Word addr is same as left-most byte
  – Addrs must be multiples of 4 to be “word-aligned”

• Pointer arithmetic not done for you in assembly
  – Must take data size into account yourself

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Assume here addr of lowest byte in word is addr of word
Data Transfer Instructions

• **Load Word** \((lw)\)
  
  – Takes data at address \(b\text{Addr}+\text{off}\) FROM memory and places it into \(\text{reg}\)

• **Store Word** \((sw)\)
  
  – Takes data in \(\text{reg}\) and stores it TO memory at address \(b\text{Addr}+\text{off}\)

• **Example Usage:**

  
  \[\begin{align*}
  \# \text{ addr of int } A[] & \rightarrow s3, \text{ a } \rightarrow s2 \\
  lw & \ t0, 12(s3) \ # \$t0=A[3] \\
  \text{add} & \ t0, s2, t0 \ # \$t0=A[3]+a \\
  sw & \ t0, 40(s3) \ # A[10]=A[3]+a
  \end{align*}\]
Great Idea #3: Principle of Locality/Memory Hierarchy

- Processor
  - Registers
- CPU
  - Processor Register
  - CPU Cache
    - Level 1 (L1) Cache
    - Level 2 (L2) Cache
    - Level 3 (L3) Cache
- Physical Memory
  - Random Access Memory (RAM)
- Solid State Memory
  - Non-volatile Flash-based Memory
- Virtual Memory
  - File-based Memory
- SSD, Flash Drive
- EDO, SD-RAM, DDR-SDRAM, RD-RAM and More...
- Mechanical Hard Drives

Categories:
- Super Fast, Super Expensive, Tiny Capacity
- Faster, Expensive, Small Capacity
- Fast, Priced Reasonably, Average Capacity
- Average Speed, Priced Reasonably, Average Capacity
- Slow, Cheap, Large Capacity
Question: Which of the following is TRUE?

(A) \text{add t0, t1, 4(t2) is valid RISCV}

(B) If RISCV halved the number of registers available, code would not be twice as fast

(C) \text{off must be a multiple of 4 for lw t0, off(s0) to be valid}

(D) \text{lw t0, off(t1) is the same as sw t1, off(t0)}
# Meet The Staff

<table>
<thead>
<tr>
<th></th>
<th>Steven</th>
<th>Nick</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Roadside Sign</strong></td>
<td>&quot;Will lecture for pomegranates&quot;</td>
<td>&quot;Hazard&quot;</td>
</tr>
<tr>
<td><strong>Greatest Weakness</strong></td>
<td>Untimely foot cramps</td>
<td>Willpower</td>
</tr>
<tr>
<td><strong>Favorite artist?</strong></td>
<td>Elohim</td>
<td>Nas (rap)</td>
</tr>
<tr>
<td><strong>Favorite Meme (of all time)</strong></td>
<td>myself</td>
<td>Arthur</td>
</tr>
</tbody>
</table>
RISCV Agenda

- Basic Arithmetic Instructions
- Comments
- x0 (zero)
- Immediates
- Data Transfer Instructions
- **Decision Making Instructions**
- Bonus: C to RISCV Practice
- Bonus: Additional Instructions
Computer Decision Making

• In C, we had *control flow*
  – Outcomes of comparative/logical statements determined which blocks of code to execute

• In RISCV, we can’t define blocks of code; all we have are *labels*
  – Defined by text followed by a colon (e.g. `main:`) and refers to the instruction that follows
  – Generate control flow by jumping to labels
  – C has these too, but they are considered bad style
Decision Making Instructions

• **Branch If Equal** \((\text{beq})\)
  
  - `beq reg1, reg2, label`
  
  - If value in `reg1` = value in `reg2`, go to label

• **Branch If Not Equal** \((\text{bne})\)
  
  - `bne reg1, reg2, label`
  
  - If value in `reg1` ≠ value in `reg2`, go to label

• **Jump** \((\text{j})\)
  
  - `j label`
  
  - Unconditional jump to label
Breaking Down the If Else

C Code:

```c
if(i==j) {
    a = b  /* then */
} else {
    a = -b /* else */
}
```

In English:

- If TRUE, execute the **THEN** block
- If FALSE, execute the **ELSE** block

RISCV (beq):

```riscv
# i→s0, j→s1
# a→s2, b→s3
beq s0,s1,???
??? This label unnecessary
sub s2, x0, s3
j end
then:
add s2, s3, x0
end:
```
Breaking Down the If Else

C Code:

```c
if(i==j) {
    a = b  /* then */
} else {
    a = -b /* else */
}
```

In English:

- If TRUE, execute the **THEN** block
- If FALSE, execute the **ELSE** block

RISCV (bne):

```
# i→s0, j→s1
# a→s2, b→s3
bne s0,s1,???
???
add s2, s3, x0
j   end
else:
sub s2, x0, s3
end:
```
Branching on Conditions other than (Not) Equal

• **Set Less Than** (*slt*)
  
  – `slt dst, reg1, reg2`
  
  – *If value in reg1 < value in reg2, dst = 1, else 0*

• **Set Less Than Immediate** (*slti*)
  
  – `slti dst, reg1, imm`
  
  – *If value in reg1 < imm, dst = 1, else 0*
Breaking Down the If Else

C Code:

```c
if(i < j) {
    a = b  /* then */
} else {
    a = -b /* else */
}
```

In English:

- If TRUE, execute the **THEN** block
- If FALSE, execute the **ELSE** block

RISCV (???):

```riscv
# i→s0, j→s1
# a→s2, b→s3
slt t0 s0 s1
??? t0,??? else
then:
add s2, s3, x0
j  end
else:
sub s2, x0, s3
end:
```
Branching on Conditions other than (Not) Equal

• Branch Less Than (blt)
  – blt reg1, reg2, label
  – If value in reg1 < value in reg2, go to label

• Branch Greater Than or Equal (bge)
  – bge reg1, reg2, label
  – If value in reg1 >= value in reg2, go to label
Breaking Down the If Else

C Code:

```c
if(i < j) {
  a = b  /* then */
} else {
  a = -b /* else */
}
```

In English:

- If TRUE, execute the **THEN** block
- If FALSE, execute the **ELSE** block

RISCV (???):

```assembly
# i→s0, j→s1
# a→s2, b→s3

??? s0,s1,else
then:
add s2, s3, x0
j  end
else:
sub s2, x0, s3
end:
```
Loops in RISCV

• There are three types of loops in C:
  – while, do...while, and for
  – Each can be rewritten as either of the other two, so the same concepts of decision-making apply

• You will examine how to write these in RISCV in discussion

• **Key Concept:** Though there are multiple ways to write a loop in RISCV, the key to decision-making is the conditional branch
Question: Which of the following is FALSE? (and if TRUE, try writing it out)

(A) We can make an unconditional branch from a conditional branch instruction

(B) We can make a for loop with just \( j \) (no \( \text{beq} \) or \( \text{bne} \))

(C) We can make a for loop without using \( j \)

(D) Not every control flow segment written with \( \text{beq} \) can be written in the same number of lines with \( \text{bne} \)
Summary

• Computers understand the *instructions* of their ISA

• RISC Design Principles
  – Smaller is faster, keep it simple

• RISCV Registers: s0–s11, t0–t6, x0

• RISCV Instructions
  – Arithmetic: add, sub, addi
  – Data Transfer: lw, sw
  – Branching: beq, bne, j, slt, slti, blt, bge

• Memory is byte-addressed
BONUS SLIDES

You are responsible for the material contained on the following slides, though we may not have enough time to get to them in lecture. You may learn the material just by doing other coursework, but hopefully these slides will help clarify the material.
Agenda

• Machine Languages
• Registers
• Administrivia
• Instructions and Immediates
• Data Transfer Instructions
• Decision Making Instructions
• Bonus:  C to RISCV Practice
• Bonus:  Additional Instructions
C to RISCV Practice

• Let’s put our all of our new RISCV knowledge to use in an example: “Fast String Copy”

• C code is as follows:

```c
/* Copy string from p to q */
char *p, *q;
while((*q++ = *p++) != '\0') ;
```

• What do we know about its structure?
  – Single `while` loop
  – Exit condition is an equality test
C to RISCV Practice

- Start with code skeleton:

```c
# copy String p to q
# p→s0, q→s1 (pointers)
Loop:
    t0 = *p
    *q = t0
    p = p + 1
    q = q + 1
    if *p==0, go to Exit
    go to Loop
```

Exit:
### C to MIPS Practice

#### Fill in lines:

```c
# copy String p to q
# p→s0, q→s1 (pointers)
Loop:  lb   t0,0(s0) # t0 = *p
       sb   t0,0(s1) # *q = t0
       addi s0,s0,1 # p = p + 1
       addi s1,s1,1 # q = q + 1
       beq  t0,0,Exit # if *p==0, go to Exit
       j Loop # go to Loop
Exit:  
```

6/25/2018
C to MIPS Practice

- Finished code:

```assembly
# copy String p to q
# p→$s0, q→$s1 (pointers)

Loop: lb  t0,0(s0)   # t0 = *p
      sb  t0,0(s1)   # *q = t0
      addi s0,s0,1   # p = p + 1
      addi s1,s1,1   # q = q + 1
      beq  t0,x0,Exit # if *p==0, go to Exit
      j    Loop      # go to Loop

Exit:   # N chars in p => N*6 instructions
```
• Alternate code using bne:

# copy String p to q
# p→s0, q→s1 (pointers)
Loop: lb   t0,0(s0)    # t0 = *p
   sb   t0,0(s1)    # *q = t0
   addi s0,s0,1    # p = p + 1
   addi s1,s1,1    # q = q + 1
   bne  t0,x0,Loop  # if *p!=0, go to Loop
# N chars in p => N*5 instructions
Agenda

• Machine Languages
• Registers
• Administrivia
• Instructions and Immediates
• Data Transfer Instructions
• Decision Making Instructions
• Bonus: C to RISCV Practice
• **Bonus: Additional Instructions**
MIPS Arithmetic Instructions

- **Multiplication** *(mul and mulh)*
  - `mul dst, src1, src2`
  - `mulh dst, src1, src2`
  - `src1*src2`: lower 32-bits through `mul`, upper 32-bits in `mulh`

- **Division** *(div)*
  - `div dst, src1, src2`
  - `rem dst, src1, src2`
  - `src1/src2`: quotient via `div`, remainder via `rem`
### RISCV Bitwise Instructions

**Note:** \(a \rightarrow s_1, b \rightarrow s_2, c \rightarrow s_3\)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>C</th>
<th>RISCV</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>(a = b &amp; c;)</td>
<td>and (s_1, s_2, s_3)</td>
</tr>
<tr>
<td>And Immediate</td>
<td>(a = b &amp; 0x1;)</td>
<td>andi (s_1, s_2, 0x1)</td>
</tr>
<tr>
<td>Or</td>
<td>(a = b \mid c;)</td>
<td>or (s_1, s_2, s_3)</td>
</tr>
<tr>
<td>Or Immediate</td>
<td>(a = b \mid 0x5;)</td>
<td>ori (s_1, s_2, 0x5)</td>
</tr>
<tr>
<td>Exclusive Or</td>
<td>(a = b ^ c;)</td>
<td>xor (s_1, s_2, s_3)</td>
</tr>
<tr>
<td>Exclusive Or Immediate</td>
<td>(a = b ^ 0xF;)</td>
<td>xori (s_1, s_2, 0xF)</td>
</tr>
</tbody>
</table>
Shifting Instructions

• In binary, shifting an unsigned number left is the same as multiplying by the corresponding power of 2
  – Shifting operations are faster
  – Does not work with shifting right/division

• *Logical shift*: Add zeros as you shift

• *Arithmetic shift*: Sign-extend as you shift
  – Only applies when you shift right (preserves sign)

• Shift by immediate or value in a register
# Shifting Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift Left Logical</td>
<td>sll \ s1,s2,s3</td>
</tr>
<tr>
<td>Shift Left Logical Imm</td>
<td>slli \ s1,s2,imm</td>
</tr>
<tr>
<td>Shift Right Logical</td>
<td>srl \ s1,s2,s3</td>
</tr>
<tr>
<td>Shift Right Logical Imm</td>
<td>srli \ s1,s2,imm</td>
</tr>
<tr>
<td>Shift Right Arithmetic</td>
<td>sra \ s1,s2,s3</td>
</tr>
<tr>
<td>Shift Right Arithmetic Imm</td>
<td>srai \ s1,s2,imm</td>
</tr>
</tbody>
</table>

- When using immediate, only values 0-31 are practical
- When using variable, only lowest 5 bits are used (read as unsigned)
Shifting Instructions

# sample calls to shift instructions
addi    t0,x0 ,-256  # t0=0xFFFFFFFFF00
slli    s0,t0,3     # s0=0xFFFFF800
srli    s1,t0,8     # s1=0x00FFFFFF
srai    s2,t0,8     # s2=0xFFFFFFFF

addi    t1,x0 ,-22   # t1=0xFFFFFFFFFEA
             # low 5: 0b01010
sll      s3,t0,t1   # s3=0xFFFFC0000
# same as slli s3,t0,10
Shifting Instructions

• Example 1:

# lb using lw:  lb s1,1(s0)
lw     s1,0(s0)  # get word
andi   s1,s1,0xFF00 # get 2\textsuperscript{nd} byte
srli   s1,s1,8   # shift into lowest
Shifting Instructions

• Example 2:

# sb using sw:  sb s1,3(s0)
lw  t0,0(s0)  # get current word
andi  t0,t0,0xFFFFFFFF # zero top byte
slli  t1,s1,24  # shift into highest
or  t0,t0,t1  # combine
sw  t0,0(s0)  # store back
Shifting Instructions

• Extra for Experts:
  – Rewrite the two preceding examples to be more general
  – Assume that the byte offset (e.g. 1 and 3 in the examples, respectively) is contained in $s_2$

• Hint:
  – The variable shift instructions will come in handy
  – Remember, the offset can be negative