Course overview

What is the course about?

- This course is about
  - how a computer works
  - how a computer is designed

- Course overview
- Introduction to ISA

Overview

• Course overview
• Introduction to ISA

COMP3211/9211 2016S1 wk1_1 P3

What is this course about?

• This course is about
  - how a computer works
  - how a computer is designed

COMP3211/9211

Overview

• Course overview
• Introduction to ISA

COMP3211/9211

What is the course about?

- Course overview
- Introduction to ISA

COMP3211/9211

Overview

• Course overview
• Introduction to ISA

COMP3211/9211
What is the course about? (cont.)

- **Computer system hierarchical abstraction**
  - Application software
    - Written in high-level language
  - System software
    - Compiler: translates HLL code to machine code
    - Operating System: service code
      - Handling input/output
      - Managing memory and storage
      - Scheduling tasks & sharing resources
  - Hardware
    - Processor, memory, I/O controllers

- **Levels of Program Code**
  - High-level language
    - Level of abstraction closer to problem domain
    - Provides for productivity and portability
  - Assembly language
    - Textual representation of instructions
  - Hardware representation
    - Binary digits (bits)
    - Encoded instructions and data

**Inside the Processor**

- **AMD Barcelona: 4 processor cores**
What is the course about? (cont.)

- Computer system hierarchical abstraction

What is computer architecture?

- Frederick P. Brooks:
  - Computer architecture, like other architecture, is the art of determining the needs of the user of a structure and then designing to meet those needs as effectively as possible within economic and technological constraints.

What is computer architecture?

- Computer architecture includes
  - ISA (instruction set architecture)
  - Machine organization
- ISA is the interface between hw and sw of a computer system. It provides attributes visible to the programmer and specifications of functions to be implemented by hardware
  - e.g. Is there a multiply instruction?
- Machine organization is how features are implemented
  - e.g. Is there a hardware multiply unit or is the multiplication done by repeated additions?

ISA & Organization

- A family of machines can often share the basic architecture
  - To provide code compatibility
    - E.g. Intel x86 family
- However, organization varies significantly between versions

- Modern instruction set architectures:
  - IA-32, PowerPC, MIPS, SPARC, ARM, and others

38 x 25 = ?
Aims of this course

• Develop a deeper understanding of computer systems
  – as a foundation for lifelong study of computer systems design

• Know how to verify and evaluate a design
  – Appreciate the importance of simulation as the primary means of validating designs and assessing performance

• Practice professional skills in design and analysis, project management, and presentation

How to achieve them

• Develop a deeper understanding of computer systems design
  – Learn general principles and historical perspectives
    – What, how and why?
    – Based on a popular RISC architecture
  – Hands-on exercises
    – Experiencing typical design phases
      – Problem analysis, solution development, modeling...

• Know how to verify and evaluate a design
  – Appreciate the importance of simulation as the primary means of validating designs and assessing performance
    – Hardware modeling and Xilinx/Modelsim simulation tools

• Practice professional skills in design and analysis, project management, and presentation
  – Group project work
  – Discussions in tutorial classes

How is the course organized?

• Lectures
  • ISA design
  • Processor
  • Memory hierarchy
  • Bus system
  • Multi-processor system
  • Advanced design topics

• Tutorials/Labs
  – Discussion, project work, presentation

Lectures

• Approximate schedule:
  – 1 week: ISA design
  – 1 week: single cycle datapath design
  – 0.5 week: multicycle datapath
  – 2.0 weeks: Pipelining
  – 1.5 weeks: memory hierarchy
  – 1.5 weeks: bus system and multi-processor system
  – 1.5 weeks: advanced design topics
  – 1 week: review
Tutorials & Labs

• One hour tut and two hour lab, per week
  – Tute 2-12
  – Lab 2-12
• Participation is essential
  – To enhance your understanding of course material
  – To effectively communicate with your project partners as well as other fellow students
  – To complete the project work; and
  – To present your project work

Tutorials

• 1 hour
• Exercises will be released by Friday of the previous week before the tutorial
  – You are required to attempt the questions before the tute class

Labs

• 2 hours
• We will use Xilinx WebPACK for designing/modelling, and ModelSIM for simulating computer components and systems –
  – these are available for download but must register on-line

Project

• Tasks 1 & 2
  – Done in a group of 3 people
    • Formed by week 2.
    • Register your group with your lab tutor
• Task 3
  – Done individually
• Project will involve presentation and lab demonstration
Quiz

• There will be a 50 minute quiz held in Week 6
• The quiz will cover all lecture, tutorial, and lab materials covered in weeks 1-4

Expectations

• Lecture
  – Design principles
    • What are they?
    • Why so?
  – Performance
    • Concepts and evaluation approaches
  – Single-cycle, multiple-cycle, pipelined datapath design
    • Related design issues
    • How are they designed?

• Lecture (cont.)
  – Memory hierarchy
    • Why do we need it?
    • The typical design issues and related solutions
  – Buses
    • How does a bus system affect the overall performance?
    • Typical bus structures
  – Multiprocessor system
    • What design issues?

• Lab
  – Lab equipment and tools
    • VHDL modeling and simulation
      – Using Xilinx WebPACK and ModelSIM
  – Completion of all labs
    • Prepare before lab
      – Understand the problems
      – Get familiar with the tools
      – Finish tasks

• Expectations

• Lab equipment and tools
  • VHDL modeling and simulation
    • Using Xilinx WebPACK and ModelSIM
  • Completion of all labs
    • Prepare before lab
      • Understand the problems
      • Get familiar with the tools
      • Finish tasks

• Lab
  • Memory hierarchy
    • Why do we need it?
    • The typical design issues and related solutions
  • Buses
    • How does a bus system affect the overall performance?
    • Typical bus structures
  • Multiprocessor system
    • What design issues?
Expectations

• Tutorial
  – Attempt all questions before tutorial
  – Participate in the tute class
    • Discuss and understand solutions

• Project
  – Be responsible for your task in the project
  – Be responsible for the progress of the whole group project
  – Try to improve your presentation skills and teamwork skills through project work
  – Participate evaluation of the project work done by other groups

Assessments

• Project
  – 35% total
    • Task 1 presentation, 7%
    • Task 2 presentation, 7%
    • Task 3 demonstration and report, 7%
    • Tasks 1&2 project report, 14%
  
• Teamwork performance
  – 5%
    • based on lab performance.
    • participation of task 1 and 2.

• There may be some adjustments if members in a group have very unbalanced contributions

• Quiz
  – 20%

• Final exam
  – 2 hours
  – 40%

• Final result = Participation + Project + Quiz + Exam
  – To pass the course, you must have (final result >= 50) && (final exam >= 40)
Staff

- Annie Guo
  - K17-501F, Ext 57136, huig@cse.unsw.edu.au
  - Consultation: Fridays, 3-5pm
- Mahanama
  - mahanamaw@cse.unsw.edu.au

Textbook and References

- Textbook
- Reference books and docs
  - See the course website for a list of references and downloadable software
- Lecture notes
  - Posted each week before the lecture
  - The lecture slides may not be the same as the lecture notes. You are encouraged to take notes.

Support

- Course website
  - http://www.cse.unsw.edu.au/~cs3211
- Course staff
  - The staff contact information can be found on course website
  - Note: email from your non-CSE account may not be answered

Introduction to Instruction Set Architecture and Design

Lecturer: Dr. Hui Annie Guo
huig@cse.unsw.edu.au
K17-501F (ext. 57136)
Introduction to ISA

- What is instruction set architecture?
- Four instruction set design principles
- Typical design issues and guidelines
  - Registers
  - Addressing mode
  - Instruction format
  - Operations
  - Data
- MIPS instruction set

ISA - Interface between HW/SW

Levels of Representation

<table>
<thead>
<tr>
<th>High Level Language Program</th>
<th>Assembly Language Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>Assembler</td>
</tr>
<tr>
<td>Machine Language Program</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine Interpretation</td>
<td></td>
</tr>
<tr>
<td>Control Signal Specification</td>
<td></td>
</tr>
</tbody>
</table>

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

w $15, 0($2)
lw $15, 0($2)
sw $15, 0($2)

ALUOP[0:3] <= InstReg[9:11] & MASK

Execution Cycle

Obtain instruction from program storage

Determine required actions

Locate and obtain operand data

Compute result value or status

Deposit results in storage for later use

Determine next instruction to be executed
Von Neumann Machine

Instruction Set Architecture: What Must be Specified?

- Operations
  - What basic functions are supported?
- Data type and size
- Operands
  - How many and where to find
- Instruction format and encoding
  - How to represent an instruction?
- Next instruction
  - Jumps, conditions, branches
  - fetch-decode-execute is implicit!

Four Basic ISAs

- Accumulator-based (1 register)
  - 1 address: add A acc acc + mem[A]
  - 1+x address: addx A acc acc + mem[A + x]
- Stack-based:
  - 0 address: add tos tos + next
- General Purpose Register: Register/Memory
  - 2 address: add A B EA(A) EA(A) + EA(B)
  - 3 address: add A B C EA(A) EA(B) + EA(C)
- General Purpose Register: Load/Store
  - 3 address: add Ra Rb Rc Ra Rb + Rc
  - 2 address: load Ra Rb Ra mem[Rb] store Ra Rb mem[Rb] Ra

Programming examples for four ISA classes

- Code sequence for C = A + B
  - Assume A, B, C are in memory

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register (register-memory)</th>
<th>Register (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1,A</td>
<td>Load R1,A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R1,B</td>
<td>Load R2,B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store R1,C</td>
<td>Add R3,R1,R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td>Store R3, C</td>
</tr>
</tbody>
</table>
Instruction Set Architecture Design

- **Goal:**
  - The instruction set should be easy to implement. It should give good performance, and possibly more

- **Four (instruction set) design principles:**
  - Smaller is faster
  - Simplicity favors regularity
  - Make the common case fast
  - Good design demands a compromise

Example of Four Design Principles: MIPS

- **Simple (Simplicity favors regularity)**
  - MIPS instructions are all 32-bits large
  - Arithmetic instructions always have three operands
  - Operands of arithmetic instructions are in registers

- **Small (Smaller is faster)**
  - MIPS has a small register file of only 32 registers, each with 32 bits

- **Compromise (Good design demands a compromise)**
  - MIPS has three instruction formats

- **Optimizing of common case (making a common occurrence fast)**
  - Immediate values are provided in I-type instructions

Variety of ISAs

- **There are many different ISAs**
- **We focus general-purpose register machines in this course**

Introduction to ISA

- **What is instruction set architecture?**
- **Basic components of an instruction set**
  - Instruction formats
  - Addressing modes
- **Four instruction set design principles**
- **Typical design issues in ISA design and guidelines**
  - Registers
  - Addressing mode
  - Instruction format
  - Operations
  - Data
- **MIPS instruction set**
Example: \( C = A + B \)

- **Accumulator-based (1 register)**
  - 1 address
  - add A
  - acc acc(B) + mem[A]

- **Stack-based:**
  - 0 address
  - add
tos tos + next

- **General Purpose Register: Load/Store**
  - 2 address
  - load Ra Mem(A) Ra mem[A]
  - load Rb Mem(B) Rb mem[B]
  - add Ra Rb A A+B
  - store Ra Mem(A) mem[A] Ra

---

**General Purpose Registers**

- Since mid 1970’s all machines use general purpose registers
- Advantages
  - registers are faster than memory
  - registers are easier for a compiler to use
    - e.g., \((A*B) - (C*D) - (E*F)\), multiplications can be done in any order, whereas, with stack they can’t.
  - registers can hold variables
    - memory traffic is reduced, so program is sped up, (since registers are faster than memory)
    - increase throughput since one instruction can manipulate multiple registers
    - code density improves (since register named with fewer bits than memory location)

---

**MIPS Integer Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero constant 0</td>
</tr>
<tr>
<td>1</td>
<td>at reserved for assembler</td>
</tr>
<tr>
<td>2</td>
<td>v0 expression evaluation &amp;</td>
</tr>
<tr>
<td>3</td>
<td>v1 function results</td>
</tr>
<tr>
<td>4</td>
<td>a0 arguments</td>
</tr>
<tr>
<td>5</td>
<td>a1</td>
</tr>
<tr>
<td>6</td>
<td>a2</td>
</tr>
<tr>
<td>7</td>
<td>a3</td>
</tr>
<tr>
<td>8</td>
<td>t0 temporary: caller saves</td>
</tr>
<tr>
<td></td>
<td>... (callee can clobber)</td>
</tr>
<tr>
<td>16</td>
<td>s0 callee saves</td>
</tr>
<tr>
<td></td>
<td>... (caller can clobber)</td>
</tr>
<tr>
<td>17</td>
<td>s1</td>
</tr>
<tr>
<td>18</td>
<td>t8 temporary (cont’d)</td>
</tr>
<tr>
<td>19</td>
<td>t9</td>
</tr>
<tr>
<td>20</td>
<td>k0 reserved for OS kernel</td>
</tr>
<tr>
<td>21</td>
<td>k1</td>
</tr>
<tr>
<td>22</td>
<td>gp Pointer to global area</td>
</tr>
<tr>
<td>23</td>
<td>sp Stack pointer</td>
</tr>
<tr>
<td>24</td>
<td>fp frame pointer</td>
</tr>
<tr>
<td>25</td>
<td>ra Return Address (HW)</td>
</tr>
</tbody>
</table>

---

**Memory Addressing**

- Since 1980 almost every machine uses byte addresses
- Two issues of multi-byte objects stored in memory:
  - Endianness
    - The order of the multiple bytes stored in the memory
  - Alignment
    - The boundary of the multi-byte object in the memory
Addressing Objects: Endianess and Alignment

- **Big Endian:**
  - address of most significant byte = word address
  - Most significant byte stored first
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- **Little Endian:**
  - address of least significant byte = word address
  - Least significant byte stored first
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

- **Alignment:**
  - objects fall on address that is multiple of their size.

---

Possible Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Add R4,#3</td>
<td>R4 ← R4+3</td>
</tr>
<tr>
<td>Register</td>
<td>Add R4,R3</td>
<td>R4 ← R4+R3</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>R1 ← R1+Mem[1001]</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Add R4,(R1)</td>
<td>R4 ← R4+Mem[R1]</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R4,100(R1)</td>
<td>R4 ← R4+Mem[100+R1]</td>
</tr>
<tr>
<td>Indexed / Base</td>
<td>Add R3,(R1+R2)</td>
<td>R3 ← R3+Mem[R1+R2]</td>
</tr>
<tr>
<td>Auto-increment</td>
<td>Add R1,(R2)+</td>
<td>R1 ← R1+Mem[R2]; R2 ← R2+d</td>
</tr>
<tr>
<td>Auto-decrement</td>
<td>Add R1,(R2)</td>
<td>R2 ← R2−d; R1 ← R1+Mem[R2]</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>R1 ← R1+Mem[100+R2+R3*d]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1,@(R3)</td>
<td>R1 ← R1+Mem[Mem[R3]]</td>
</tr>
</tbody>
</table>

---

Which to use?

- Analysis and profiling.

---

Addressing Mode Usage?

3 programs measured on machine with all address modes (VAX)

- Displacement: 42% avg, 32% to 55%
- Immediate: 33% avg, 17% to 43%
- Register indirect: 13% avg, 3% to 24%
- Scaled: 7% avg, 0% to 16%
- Memory indirect: 3% avg, 1% to 6%
- Misc: 2% avg, 0% to 3%

75% displacement & immediate
88% displacement, immediate & register indirect

What displacement values need to be supported?
What immediate values can be expected?
**Displacement Address Size?**

- Avg. of 5 SPECint92 programs vs. avg. 5 SPECfp92 programs
- 1% of addresses > 16-bits
- 12 - 16 bits of displacement needed

**Immediate Size?**

- 50% to 60% fit within 8 bits
- 75% to 80% fit within 16 bits

**Addressing Modes Guideline Adopted by MIPS**

- **Displacement**
  - Displacement size should be 12 to 16 bits

- **Immediate**
  - Immediate size should be 8 to 16 bits

- **Register Indirect**

**Generic Examples of Instruction Format Widths**

- **Variable:**
  
- **Fixed:**
  
- **Hybrid:**
Some Instruction Format Design Strategies

- If code size is most important (as in some embedded apps), use variable length instructions
- If performance is most important, use fixed length instructions as MIPS does
- Embedded machines (ARM, MIPS) added optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16)
  - decide performance or density for each procedure
- If you have many memory operands per instruction and many addressing modes
  - use an Address Specifier per operand (VAX)
- If you have load-store machine with at most 1 address per instruction and only one or two addressing modes

MIPS Instruction Set Architecture

- We’ll be working with the MIPS instruction set architecture
  - A typical architecture since 1980's
  - millions MIPS processors manufactured in just 2002
  - used by ATI Technologies, Broadcom, NEC, Nintendo, Cisco, Silicon Graphics, Sony, ...

Features of MIPS ISA

- Instructions, all of 32 bits (3 formats)
- 32 x 32-bit GPR (R0 contains zero) and 32 FP registers (HI and LO)
  - partitioned by software convention
- 3-address, reg-reg arithmetic instructions
  - AL operations are always performed on registers
- Single address mode for load/store:
  - base+displacement
- Simple branch conditions
  - compare against zero or two registers
  - no integer condition codes

5 MIPS Addressing Modes/3 Instruction Formats

- 1. Register (direct) (e.g. addition)
- 2. Immediate (e.g. increment)
- 3. Base+index (e.g. array access)
- 4. PC-relative (e.g. branch)
- 5. Pseudodirect (e.g. jump)
### MIPS Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add</code></td>
<td><code>add $1,$2,$3</code></td>
<td>$1 = $2 + $3`</td>
<td>3 operands;</td>
</tr>
<tr>
<td><code>subtract</code></td>
<td><code>sub $1,$2,$3</code></td>
<td>$1 = $2 – $3`</td>
<td>3 operands;</td>
</tr>
<tr>
<td><code>add immediate</code></td>
<td><code>addi $1,$2,100</code></td>
<td>$1 = $2 + 100`</td>
<td>+ constant;</td>
</tr>
<tr>
<td><code>add unsigned</code></td>
<td><code>addu $1,$2,$3</code></td>
<td>$1 = $2 + $3`</td>
<td>3 operands;</td>
</tr>
<tr>
<td><code>subtract unsigned</code></td>
<td><code>subu $1,$2,$3</code></td>
<td>$1 = $2 – $3`</td>
<td>3 operands;</td>
</tr>
<tr>
<td><code>add imm. unsigned</code></td>
<td><code>addiu $1,$2,10</code></td>
<td>$1 = $2 + 10`</td>
<td>+ constant;</td>
</tr>
<tr>
<td><code>multiply</code></td>
<td><code>mult $2,$3</code></td>
<td>Hi, Lo = $2 x $3`</td>
<td>64-bit signed product</td>
</tr>
<tr>
<td><code>multiply unsign.</code></td>
<td><code>multu $2,$3</code></td>
<td>Hi, Lo = $2 x $3`</td>
<td>64-bit unsigned product</td>
</tr>
<tr>
<td><code>divide</code></td>
<td><code>div $2,$3</code></td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3`</td>
<td>Unsigned quotient &amp; remainder</td>
</tr>
<tr>
<td><code>divide unsign.</code></td>
<td><code>divu $2,$3</code></td>
<td>Lo = $2 ÷ $3, Hi = $2 mod $3`</td>
<td>Unsigned quotient &amp; remainder</td>
</tr>
<tr>
<td><code>Move from Hi</code></td>
<td><code>mfhi $1</code></td>
<td>$1 = Hi`</td>
<td>Used to get copy of Hi</td>
</tr>
<tr>
<td><code>Move from Lo</code></td>
<td><code>mflo $1</code></td>
<td>$1 = Lo`</td>
<td>Used to get copy of Lo</td>
</tr>
</tbody>
</table>

### MIPS Logical Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and</code></td>
<td><code>and $1,$2,$3</code></td>
<td>$1 = $2 &amp; $3`</td>
<td>3 reg. operands; Logical AND</td>
</tr>
<tr>
<td><code>or</code></td>
<td><code>or $1,$2,$3</code></td>
<td>$1 = $2</td>
<td>$3`</td>
</tr>
<tr>
<td><code>xor</code></td>
<td><code>xor $1,$2,$3</code></td>
<td>$1 = $2 @ $3`</td>
<td>3 reg. operands; Logical XOR</td>
</tr>
<tr>
<td><code>nor</code></td>
<td><code>nor $1,$2,$3</code></td>
<td>$1 = ~(#2</td>
<td>$3)`</td>
</tr>
<tr>
<td><code>and immediate</code></td>
<td><code>andi $1,$2,10</code></td>
<td>$1 = $2 &amp; 10`</td>
<td>Logical AND reg, constant</td>
</tr>
<tr>
<td><code>or immediate</code></td>
<td><code>ori $1,$2,10</code></td>
<td>$1 = $2</td>
<td>10`</td>
</tr>
<tr>
<td><code>xor immediate</code></td>
<td><code>xori $1,$2,10</code></td>
<td>$1 = ~$2 &amp;~10`</td>
<td>Logical XOR reg, constant</td>
</tr>
<tr>
<td><code>shift left logical</code></td>
<td><code>sll $1,$2,10</code></td>
<td>$1 =$2 &lt;&lt; 10`</td>
<td>Shift left by constant</td>
</tr>
<tr>
<td><code>shift right logical</code></td>
<td><code>srl $1,$2,10</code></td>
<td>$1 =$2 &gt;&gt; 10`</td>
<td>Shift right by constant</td>
</tr>
<tr>
<td><code>shift right arithm.</code></td>
<td><code>sra $1,$2,10</code></td>
<td>$1 =$2 &gt;&gt; 10`</td>
<td>Shift right (sign extend)</td>
</tr>
<tr>
<td><code>shift left logical</code></td>
<td><code>slv $1,$2,$3</code></td>
<td>$1 =$2 &lt;&lt; $3`</td>
<td>Shift left by variable</td>
</tr>
<tr>
<td><code>shift right logical</code></td>
<td><code>sr $1,$2,$3</code></td>
<td>$1 =$2 &gt;&gt; $3`</td>
<td>Shift right by variable</td>
</tr>
<tr>
<td><code>shift right arithm.</code></td>
<td><code>srav $1,$2,$3</code></td>
<td>$1 =$2 &gt;&gt; $3`</td>
<td>Shift right arith. by variable</td>
</tr>
</tbody>
</table>

### MIPS Data Transfer Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SW R3, 500(R4)</code></td>
<td>Store word</td>
<td>Mem[R4 + 500] ← R3</td>
</tr>
<tr>
<td><code>SH R3, 502(R2)</code></td>
<td>Store half</td>
<td></td>
</tr>
<tr>
<td><code>SB R2, 41(R3)</code></td>
<td>Store byte</td>
<td></td>
</tr>
<tr>
<td><code>LW R1, 30(R2)</code></td>
<td>Load word</td>
<td>R1 ← Mem[R2 + 30]</td>
</tr>
<tr>
<td><code>LH R1, 40(R3)</code></td>
<td>Load halfword</td>
<td></td>
</tr>
<tr>
<td><code>LHU R1, 40(R3)</code></td>
<td>Load halfword unsigned</td>
<td></td>
</tr>
<tr>
<td><code>LB R1, 40(R3)</code></td>
<td>Load byte</td>
<td></td>
</tr>
<tr>
<td><code>LBU R1, 40(R3)</code></td>
<td>Load byte unsigned</td>
<td></td>
</tr>
<tr>
<td><code>LUI R1, 40</code></td>
<td>Load Upper Immediate (16 bits shifted left by 16)</td>
<td></td>
</tr>
</tbody>
</table>

### MIPS jump, branch, compare Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>beq $1,$2,100</code></td>
<td>if ($1 == $2) go to PC+4+100</td>
<td>Equal test; PC relative branch</td>
</tr>
<tr>
<td><code>bne $1,$2,100</code></td>
<td>if ($1 != $2) go to PC+4+100</td>
<td>Not equal test; PC relative</td>
</tr>
<tr>
<td><code>slt $1,$2,100</code></td>
<td>if ($2 &lt; $3) $1=1; else $1=0</td>
<td>Compare less than; 2's comp.</td>
</tr>
<tr>
<td><code>slti $1,$2,100</code></td>
<td>if ($2 &lt; 100) $1=1; else $1=0</td>
<td>Compare &lt; constant; 2's comp.</td>
</tr>
<tr>
<td><code>sltiu $1,$2,100</code></td>
<td>if ($2 &lt; $3) $1=1; else $1=0</td>
<td>Compare &lt; constant; natural numbers</td>
</tr>
<tr>
<td><code>j 10000</code></td>
<td>go to 10000</td>
<td>Jump to target address</td>
</tr>
<tr>
<td><code>jr $31</code></td>
<td>go to $31</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td><code>jal 10000</code></td>
<td>$31 = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
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

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

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $t0, $t1, ..., $t9 for temporary values
  - $s0, $s1, ..., $s7 for saved variables
- Design Principle 2: Smaller is faster
  - c.f. main memory: millions of locations

Register Operand Example

- C code:
  \[ f = (g + h) - (i + j); \]
  - f, ..., j in $s0, ..., $s4
- Compiled MIPS code:
  add $t0, $s1, $s2
  add $t1, $s3, $s4
  sub $s0, $t0, $t1
Memory Operands

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

Memory Operand Example 1

- C code:
  \[ g = h + A[8]; \]
  - \( g \) in \$s1, \( h \) in \$s2, base address of \( A \) in \$s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word
  \[
  lw \quad t0, 32(s3) \quad # \text{load word}
  add \quad s1, s2, t0
  \]

Memory Operand Example 2

- C code:
  \[ A[12] = h + A[8]; \]
  - \( h \) in \$s2, base address of \( A \) in \$s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
  \[
  lw \quad t0, 32(s3) \quad # \text{load word}
  add \quad t0, s2, t0
  sw \quad t0, 48(s3) \quad # \text{store word}
  \]

Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!
Immediate Operands

• Constant data specified in an instruction
  addi $s3, $s3, 4
• No subtract immediate instruction
  – Just use a negative constant
    addi $s2, $s1, -1
• **Design Principle 3: Make the common case fast**
  – Small constants are common
  – Immediate operand avoids a load instruction

The Constant Zero

• MIPS register 0 ($zero) is the constant 0
  – Cannot be overwritten
• Useful for common operations
  – E.g., move between registers
    add $t2, $s1, $zero

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

32-bit Constants

• Most constants are small
  – 16-bit immediate is sufficient
• For the occasional 32-bit constant
  \[
  \text{lui } rt, \text{ constant}
  \]
  – Copies 16-bit constant to left 16 bits of `rt`
  – Clears right 16 bits of `rt` to 0

\[
\text{lhi } s0, 61 \\
\text{ori } s0, s0, 2304
\]
MIPS Instruction Set

- Refer to inside jacket of textbook, and Ch 2

Fallacies

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Fallacies

- Backward compatibility ⇒ instruction set doesn’t change
  - But they do accrete more instructions

Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped
Concluding Remarks

• Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises

• Layers of software/hardware
  – Compiler, assembler, hardware

• MIPS: typical of RISC ISAs
  – c.f. x86

Measure MIPS instruction executions in benchmark programs
  – Consider making the common case fast
  – Consider compromises

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>MIPS examples</th>
<th>SPEC2006 Int</th>
<th>SPEC2006 FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add, sub, addi</td>
<td>16%</td>
<td>48%</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw, sw, lb, lbu, lh, lh, sb, ltu</td>
<td>35%</td>
<td>36%</td>
</tr>
<tr>
<td>Logical</td>
<td>and, or, nor, andi, orl, slt, srl</td>
<td>12%</td>
<td>4%</td>
</tr>
<tr>
<td>Cond. Branch</td>
<td>beq, bne, slt, slti, sltui, sltu</td>
<td>34%</td>
<td>8%</td>
</tr>
<tr>
<td>Jump</td>
<td>j, jr, jal</td>
<td>2%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Homework

• Browse the course website
• Skim through textbook
• Find your project partners
• Read Chap. 2