CS G22.2233  
Computer Systems Design  
Spring 2007  

Lecture 01: Introduction  
Mohammad Banikazemi

[Slides from Prof. Mary Jane Irwin, PSU Adapted from Computer Organization and Design, Patterson & Hennessy, © 2005, UCB]

Course Administration

- Instructor: Mohammad Banikazemi  
  mb@cs.nyu.edu  
  Adjunct Professor, NYU  
  Research Staff Membrer,  
  IBM T. J. Watson Research Center  
  401 WWH Building  
  Office Hrs: W 7:00-8:00pm

- Text: Required: Computer Org and Design,  
  3rd Edition, Patterson and Hennessy ©2005

- URL: www.cs.nyu.edu/courses/spring07/G22.2233-001/  
- Slides: pdf on the course web page after lecture

Let’s Get Started

- Motivation
- Course organization
- Administrative stuff  
  ● workload and grading
- Brief overview of SimpleScalar toolset
- Introduction to Computer Design  
  ● Target markets  
  ● Technology trends
- Instruction Set Architecture

[Patterson/Hennessy COD: HSI (3rd Edition): Chapters 1 and 2]

PC Motherboard Closeup

From the Intel 386 to the Pentium 4

Intel 386, introduced 1985  
275,000 transistors, 1 micron  
16 MHz clock speed

Intel Pentium III, introduced 1999  
5.5 M transistors, 0.25 micron  
600 MHz clock speed

Intel 4 Prescott, introduced late 04  
1.25M transistors, 0.09 micron  
2.8-3.8 GHz clock speed

Intel Pentium III Microarchitecture

IFU: Instruction fetch unit  
ID: Instruction dispatch  
MD: Micro-instruction sequencer  
BTB: Branch target buffer  
ROT: Register alias table  
RS: Reservation station  
PEU: Speculation unit  
FIC: FP execution unit  
DCU: Data cache unit  
MOB: Memory Ordering Buffer  
MIU: Mem Interface unit
How Do the Pieces Fit Together?

- Coordination of many levels of abstraction
- Under a rapidly changing set of forces
- Design, measurement, and evaluation

Course Content

- Content
  - Principles of computer architecture: CPU datapath and control unit design (single-issue pipelined, superscalar, VLIW), memory hierarchies and design, I/O organization and design, and introduction to advanced processor design (multiprocessors and SMT)
- Course goals
  - To learn the organizational paradigms that determine the capabilities and performance of computer systems. To understand the interactions between the computer’s architecture and its software so that future software designers (compiler writers, operating system designers, database programmers, ...) can achieve the best cost-performance trade-offs and so that future architects understand the effects of their design choices on software applications.

Course Structure

- Design focused class
  - Various homework assignments throughout the semester
  - Simulation of architecture alternatives using SimpleScalar
- Lectures:
  - 3-4 weeks review of the MIPS ISA and basic architecture
  - 4-5 weeks pipelined datapath design issues and superscalar
  - 2 weeks memory hierarchies and memory design issues
  - 2 weeks I/O design issues
  - 1 week introduction to multiprocessor design issues
  - 1 week exams

What You Should Be Familiar With

- Basic logic design & machine organization
  - Bits, gates, combinational and sequential logic
- Create, assemble, run, debug small programs in an assembly language
- Create, compile, and run C programs
- Create, organize, and edit files and run programs on Unix/Linux

Course Workload

- Lectures (Reading assignments from text)
- Lab and Homework assignments
  - Programming assignments
    - Building a simulator for a modern processor
    - Use this simulator to understand impact of architectural techniques
  - Homework assignments
- Midterm exam
- Final exam
  - Review and sample questions will be provided in class

Grading Information

- Grade determinates
  - Midterm Exam ~30%
    - Wed. March 21, 5:00-6:50pm, WWH 1013
  - Final Exam ~40%
    - Wed. May 2, 5:00-6:50pm, WWH 10
  - Homework and lab assignments ~30%
    - Homework assignments due at the beginning of class
    - Code to be submitted electronically by 17:00 on the due date
    - No late assignments will be accepted.

http://www.cs.nyu.edu/web/Academic/Graduate/academic_integrity.html
SimpleScalar Toolset
- Comprehensive collection of tools for evaluating new architectural techniques
  - Possible to define new instruction-set architectures (support for Alpha, PISA)
  - Modules for writing own execution-driven simulators
    - bpred, caches, statistics collection, program loading, functional unit construction
  - Many papers at recent architecture conferences use SimpleScalar

SimpleScalar Toolset (cont’d)
- For the course assignments we will be using a small subset of the tools
  - You will be using an instructional ISA called PISA
    - Closely resembles MIPS 64
  - PISA executables produced using GNU cross-compiler tools
- Goal of the assignments: To understand the issues involved in implementing and to assess the potential benefits from architectural techniques used in modern-day microprocessors
  - E.g., Branch prediction in modern-day microprocessors
    - At what stage during instruction execution is branch prediction used?
      - It takes some time to figure out that an instruction is a branch
    - How should the branch predictor be updated with information about seen branches?
    - What impact does prediction have on performance?

Outline
- Motivation
- Course organization
- Administrative stuff
  - workload and grading
- Brief overview of SimpleScalar toolset
- Introduction to Computer Design
  - Target markets
  - Technology trends
- Instruction Set Architecture

Example Machine Organization
- Workstation design target
  - 25% of cost on processor
  - 25% of cost on memory (minimum memory size)
  - Rest on I/O devices, power supplies, box

Moore’s Law
- In 1965, Gordon Moore predicted that the number of transistors that can be integrated on a die would double every 18 to 24 months (i.e., grow exponentially with time).
- Amazingly visionary – million transistor/chip barrier was crossed in the 1980’s.
  - 2300 transistors, 1 MHz clock (Intel 4004) - 1971
  - 16 Million transistors (Ultra Sparc III)
  - 42 Million transistors, 2 GHz clock (Intel Xeon) – 2001
  - 55 Million transistors, 3 GHz, 130nm technology, 250mm² die (Intel Pentium 4) - 2004
  - 140 Million transistor (HP PA-8500)
**Introduction**

**Impacts of Advancing Technology**
- **Processor**
  - Logic capacity: increases about 30% per year
  - Performance: 2x every 1.5 years
- **Memory**
  - DRAM capacity: 4x every 3 years, now 2x every 2 years
  - Memory speed: 1.5x every 10 years
  - Cost per bit: decreases about 25% per year
- **Disk**
  - Capacity: increases about 60% per year

**What is Ahead?**
- Today’s desktop microprocessors (e.g., Pentium 4 Extreme Edition)
  - 178 million transistors, 90 nanometer technology, 3.73 GHz clock speed
  - Internally: “hyper-pipelining”, multithreading, 128-bit SIMD instructions
- The future
  - Greater instruction level parallelism
  - Bigger caches, and more levels of cache
  - Multiple processors per chip (“multicore”)
  - Complete systems on a chip
  - Reducing the power consumption
  - High performance interconnects
- Breakdown into desktop, enterprise, and embedded target markets
  - Different performance criteria
- This course provides the background for you to design, analyze, and effectively use such systems

**Where Do We Start From?**

**Outline**
- Motivation
- Course organization
- Administrative stuff
  - Workload and grading
- Brief overview of SimpleScalar toolset
- Introduction to Computer Design
  - Target markets
  - Technology trends
- Instruction Set Architecture
  - [Patterson/Hennessy COD: HSI (3rd Edition): Chapters 1 and 2]

**Instruction Set Architecture (ISA)**

“Instruction set architecture is the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.”

Source: IBM in 1964 when introducing the IBM 360 architecture
- An instruction set is a functional description of processor
  - What operations can it do
  - What storage mechanisms does it support
- ISA defines the hardware/software interface
  - A good interface:
    - Lasts through many implementations
    - Can be used in many different ways
    - Provides convenient functionality to higher levels
    - Permits an efficient implementation at lower levels
Instruction Set Design Issues
- What operations are supported?
  - add, sub, mul, move, compare...
- Where are operands stored?
  - Registers (how many of them are there), memory, stack, accumulator
- How many explicit operands are there?
  - 0, 1, 2, or 3
- How is the operand location specified?
  - register, immediate, indirect, ...
- What type and size of operands are supported?
  - byte, int, float, double, string, vector, ...

RISC - Reduced Instruction Set Computer
- RISC philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited addressing modes
  - limited operations
- MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC, Intel (Compaq) Alpha, ...
- Instruction sets are measured by how well compilers use them as opposed to how well assembly language programmers use them

Design goals: speed, cost (design, fabrication, test, packaging), size, power consumption, reliability, memory space (embedded systems)

MIPS R3000 Instruction Set Architecture (ISA)
- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
  - coprocessor
  - Memory Management
  - Special

Registers
- R0 - R31

3 Instruction Formats: all 32 bits wide
- R0 - R31
- Immediate
- Jump target

MIPS Arithmetic Instructions
- MIPS assembly language arithmetic statement
  - add $t0, $s1, $s2
  - sub $t0, $s1, $s2
- Each arithmetic instruction performs only one operation
- Each arithmetic instruction fits in 32 bits and specifies exactly three operands
- Operand order is fixed (destination first)
- Those operands are all contained in the datapath's register file ($t0,$s1,$s2) - indicated by $
**MIPS Register File**

- Holds thirty-two 32-bit registers
  - Two read ports and
  - One write port
- Registers are
  - Faster than main memory
    - But register files with more locations are slower (e.g., a 64 word file could be as much as 50% slower than a 32 word file)
    - Read/write port increase impacts speed
  - Easier for a compiler to use
    - e.g., 
      \[(A+B) - (C+D) - (E*F)\] can do multplies in any order vs. stack
  - Can hold variables so that
    - code density improves (since register are named with fewer bits than a memory location)

**MIPS Memory Access Instructions**

- MIPS has two basic data transfer instructions for accessing memory
  - lw \( \$t_0, 4(\$s_3) \) #load word from memory
  - sw \( \$t_0, 8(\$s_3) \) #store word to memory
- The data is loaded into \( ((\text{lw}) \text{ or } \text{sw}) \) a register in the register file – a 5 bit address
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value
  - A 16-bit field meaning access is limited to memory locations within a region of \( 2^{12} \) or 8,192 words (\( 2^{15} \) or 32,768 bytes) of the address in the base register
  - Note that the offset can be positive or negative

**Byte Addresses**

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - The memory address of a word must be a multiple of 4 (alignment restriction)
- **Big Endian:** leftmost byte is word address
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- **Little Endian:** rightmost byte is word address
  - Intel 80x86, DEC Vax, DEC Alpha

**Aside: Loading and Storing Bytes**

- MIPS provides special instructions to move bytes
  - lb \( \$t_0, 1(\$s_3) \) #load byte from memory
  - sb \( \$t_0, 6(\$s_3) \) #store byte to memory
- What 8 bits get loaded and stored?
  - load byte places the byte from memory in the rightmost 8 bits of the destination register
    - what happens to the other bits in the register?
  - store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
    - what happens to the other bits in the memory word?
MIPS Control Flow Instructions

- MIPS conditional branch instructions:
  - `bne $s0, $s1, Lbl` # go to Lbl if $s0 ≠ $s1
  - `beq $s0, $s1, Lbl` # go to Lbl if $s0 = $s1

- Example: if (i==j) h = i + j;
  
  ```
  bne $s0, $s1, Lbl1
  add $s3, $s0, $s1
  Lbl1: ...
  ```

- Instruction Format (I format):
  ```
  op rs rt 16 bit offset
  ```

- How is the branch destination address specified?

Specify Branch Destinations

- Use a register (like in lw and sw) added to the 16-bit offset

  - which register? Instruction Address Register (the PC)
    - its use is automatically implied by instruction
    - PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction
  - limits the branch distance to $2^{16}$ instructions from the (instruction after the) branch instruction, but most branches are local anyway

- Example:
  ```
  slt $t0, $s0, $s1 # if $s0 < $s1 then
  beq $t0, $zero, Label # $s0 = $t0
  ```

More Branch Instructions

- We have `beq, bne`, but what about other kinds of branches (e.g., branch-if-less-than)?
  - For this, we need yet another instruction, `slt`

- Set on less than instruction:
  ```
  slt $t0, $s0, $s1
  ```

  - if $s0 < $s1 then
    - $t0 = 1
    - else
      - $t0 = 0

- Instruction format (R format):
  ```
  op rs rt rd funct
  ```

- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler
  - Its why the assembler needs a reserved register (`$at`)
Instructions for Accessing Procedures
- MIPS procedure call instruction: `jal ProcedureAddress` #jump and link
- Saves PC+4 in register $ra to have a link to the next instruction for the procedure return
- Machine format (I format):
  - A 26 bit address
- Then can do procedure return with a `jr $ra` #return
- Instruction format (R format):
  - op
  - rs
  - rt
  - 16 bit immediate

Aside: Spilling Registers
- What if the callee needs more registers? What if the procedure is recursive?
- Uses a stack – a last-in-first-out queue – in memory for passing additional values or saving (recursive) return address(es)
- One of the general registers, $sp, is used to address the stack (which "grows" from high address to low address)
  - add data onto the stack – push
    - $sp = $sp + 4
    - data on stack at new $sp
  - remove data from the stack – pop
    - data from stack at $sp
    - $sp = $sp - 4

Aside: How About Larger Constants?
- We’d also like to be able to load a 32 bit constant into a register, for this we must use two instructions
  - a new "load upper immediate" instruction
    - `lui $t0, 1010101010101010`
  - Then must get the lower order bits right, use
    - `ori $t0, $t0, 1010101010101010`

MIPS Immediate Instructions
- Small constants are used often in typical code
- Possible approaches?
  - put "typical constants" in memory and load them
  - create hard-wired registers (like $zero) for constants like 1
  - have special instructions that contain constants!
    - `addi $sp, $sp, 4` #$sp = $sp + 4
    - `slti $t0, $s2, 15` #$t0 = 1 if $s2<15
- Machine format (I format):
  - op
  - rs
  - rt
  - 16 bit immediate
- The constant is kept inside the instruction itself!
  - Immediate format limits values to the range $2^{15}$-1 to $-2^{15}$

Aside: How About Larger Constants?
- We’d also like to be able to load a 32 bit constant into a register, for this we must use two instructions
  - a new "load upper immediate" instruction
    - `lui $t0, 1010101010101010`
  - Then must get the lower order bits right, use
    - `ori $t0, $t0, 1010101010101010`

MIPS ISA So Far

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Op Code</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic (R &amp; I) format</td>
<td>add</td>
<td>0 and 32</td>
<td>$t1, $s2, $s3</td>
<td>$t1 = $s2 + $s3</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>0 and 34</td>
<td>$t1, $s2, $s3</td>
<td>$t1 = $s2 - $s3</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>8</td>
<td>$t1, $s2, 6</td>
<td>$t1 = $s2 + 6</td>
</tr>
<tr>
<td></td>
<td>or immediate</td>
<td>13</td>
<td>$t1, $s2, 6</td>
<td>$t1 = $s2 + 6</td>
</tr>
<tr>
<td>Data Transfer (I format)</td>
<td>load word</td>
<td>15</td>
<td>$t1, $s2($s3)</td>
<td>$t1 = Memory[$s2+24]</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>23</td>
<td>$t1, $s2($s3)</td>
<td>Memory[$s2+24] = $t1</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>31</td>
<td>$t1, $s2($s3)</td>
<td>$t1 = Memory[$s2+25]</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>39</td>
<td>$t1, $s2($s3)</td>
<td>Memory[$s2+25] = $t1</td>
</tr>
<tr>
<td></td>
<td>load upper imm</td>
<td>35</td>
<td>$t1, $s1</td>
<td>$t1 = $s1 + 2^32</td>
</tr>
<tr>
<td></td>
<td>beq</td>
<td>4</td>
<td>$s1, $s2, 1</td>
<td>if ($s1==$s2) go to L</td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>5</td>
<td>$s1, $s2, 1</td>
<td>if ($s1!= $s2) go to L</td>
</tr>
<tr>
<td></td>
<td>slt</td>
<td>10</td>
<td>$s1, $s2, $s3</td>
<td>if ($s1&lt;$s2) $s1=1 else $s1=0</td>
</tr>
<tr>
<td></td>
<td>j</td>
<td>2</td>
<td>$t1</td>
<td>go to $t1</td>
</tr>
<tr>
<td></td>
<td>jal</td>
<td>3</td>
<td>$t1</td>
<td>go to $t1; $ra=PC+4</td>
</tr>
</tbody>
</table>

MIPS Organization So Far

- Processor
  - Register file
  - Memory
    - Head/Write address
    - 0…0000
    - 0…1000
    - 0…1000
    - Word address (binary)

- Processor
  - Instruction fetch
    - PC = PC+4
  - Instruction decode
  - Instruction execute
  - Instruction store
  - Instruction branch
Review of MIPS Operand Addressing Modes

- **Register addressing** – operand is in a register
  - `op rs rt rd funct`

- **Base (displacement) addressing** – operand is at the memory location whose address is the sum of a register and a 16-bit constant contained within the instruction
  - `op rs rt offset`

- **Immediate addressing** – operand is a 16-bit constant contained within the instruction
  - `op rs rt operand`

Review of MIPS Instruction Addressing Modes

- **PC-relative addressing** – instruction address is the sum of the PC and a 16-bit constant contained within the instruction
  - `op rs rt offset`

- **Pseudo-direct addressing** – instruction address is the 26-bit constant contained within the instruction concatenated with the upper 4 bits of the PC
  - `op jump address`

MIPS (RISC) Design Principles

- **Simplicity favors regularity**
  - fixed size instructions – 32-bits
  - small number of instruction formats
  - opcode always the first 6 bits

- **Good design demands good compromises**
  - three instruction formats

- **Smaller is faster**
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes

- **Make the common case fast**
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands

Next Lecture and Reminders

- **Next lecture**
  - MIPS ALU Review
    - Reading assignment – PH, Chapter 3

- **Reminders**
  - HW1 due next week: January 24th beginning of class