ECE/CS 552: Midterm Review
Instructor: Mikko H Lipasti
Fall 2010
University of Wisconsin-Madison
Lecture notes based on notes by Mark Hill and John P. Shen
Updated by Mikko Lipasti

Computer Architecture
- Exercise in engineering tradeoff analysis
  - Find the fastest/cheapest/power-efficient/etc. solution
  - Optimization problem with 100s of variables
- All the variables are changing
  - At non-uniform rates
  - With inflection points
  - Only one guarantee: Today’s right answer will be wrong tomorrow
- Two high-level effects:
  - Technology push
  - Application Pull

Abstraction
- Difference between interface and implementation
  - Interface: WHAT something does
  - Implementation: HOW it does so

What’s the Big Deal?
- Tower of abstraction
- Complex interfaces implemented by layers below
- Abstraction hides detail
- Hundreds of engineers build one product
- Complexity unmanageable otherwise

Performance vs. Design Time
- Time to market is critically important
- E.g., a new design may take 3 years
  - It will be 3 times faster
  - But if technology improves 50%/year
  - In 3 years $1.5^3 = 3.38$
  - So the new design is worse!
  (unless it also employs new technology)

Bottom Line
- Designers must know BOTH software and hardware
- Both contribute to layers of abstraction
- IC costs and performance
- Compilers and Operating Systems
Performance

- Time and performance: Machine A n times faster than Machine B
  - Iff Time(B)/Time(A) = n
- Iron Law: Performance = Time/program = Instructions X Cycles / Program X Instruction Time X Cycle

Performance cont’d

- Other Metrics: MIPS and MFLOPS
  - Beware of peak and omitted details
- Benchmarks: SPEC2000 (95 in text)
- Summarize performance:
  - AM for time
  - HM for rate
  - GM for ratio
- Amdahl’s Law:

Ch 2 Summary

- Basics
- Registers and ALU ops
- Memory and load/store
- Branches and jumps
- Addressing Modes

Summary: Instruction Formats

- R: opcode rs rt rd shamt function
  - 6 5 5 5 6
- I: opcode rs rt address/immediate
  - 6 5 5 16
- J: opcode addr
  - 6 26
- Instruction decode:
  - Read instruction bits
  - Activate control signals

Conclusions

- Simple and regular
  - Constant length instructions, fields in same place
- Small and fast
  - Small number of operands in registers
- Compromises inevitable
  - Pipelining should not be hindered
- Make common case fast!
- Backwards compatibility!

Basic Arithmetic and the ALU

- Number representations: 2’s complement, unsigned
- Addition/Subtraction
- Add/Sub ALU
  - Full adder, ripple carry, subtraction
  - Carry-lookahead addition
  - Logical operations
    - and, or, xor, nor, shifts
  - Overflow
**Unsigned Integers**

- \( f(b_{31}..b_0) = b_{31} \times 2^{31} + \ldots + b_1 \times 2^1 + b_0 \times 2^0 \)
- Treat as normal binary number
  
  \[
  = 1 \times 2^{31} + 1 \times 2^{31} + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^1 + 1 \times 2^0 \\
  = 128 + 64 + 16 + 4 + 1 + 1 = 213
  \]
- Max \( f(111\ldots1) = 2^{32} - 1 = 4,294,967,295 \)
- Min \( f(000\ldots0) = 0 \)
- Range \([0,2^{32}-1]\) => # values \((2^{32}-1) - 0 + 1 = 2^{32}\)

**Signed Integers**

- 2's complement
  
  \[ f(b_{31} \ldots b_0) = -b_{31} \times 2^{31} + \ldots b_1 \times 2^1 + b_0 \times 2^0 \]
- Max \( f(011\ldots1) = 2^{31} - 1 = 2147483647 \)
- Min \( f(000\ldots0) = -2^{31} = -2147483648 \) (asymmetric)
- Range \([-2^{31},2^{31}-1]\) => # values \((2^{31}-1 - -2^{31} + 1) = 2^{32}\)
- E.g. \(-6 \Rightarrow 000\ldots0110 \Rightarrow 111\ldots1001 + 1 \Rightarrow 111\ldots1010\)

**Full Adder**

- Full adder \((a,b,c_{in}) \Rightarrow (c_{out}, s)\)
- \( c_{out} = \text{two or more of } (a, b, c_{in}) \)
- \( s = \text{exactly one or three of } (a,b,c_{in}) \)

<table>
<thead>
<tr>
<th>(a)</th>
<th>(b)</th>
<th>(c_{in})</th>
<th>(c_{out})</th>
<th>(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Combined Ripple-carry Adder/Subtractor**

- Control = 1 => subtract
- XOR B with control and set \(c_{in_0}\) to control

**4-bit Carry Lookahead Adder**

**Hierarchical Carry Lookahead for 16 bits**
CLA: Compute G’s and P’s

CLA: Compute Carries

All Together

Addition Overflow

Subtraction Overflow

What to do on Overflow?

• 2 + 3 = 5 > 4; 010 + 011 = 101 =? –3 < 0
  – X is f(2)
• -1 + -4: 111 + 100 = 011 > 0
  – Y is ~f(2)
Overflow = f(2) * ~(a2) * ~(b2) + ~f(2) * a(2) * b(2)

• No overflow on a-b if signs are the same
• Neg – pos => neg ;; overflow otherwise
• Pos – neg => pos ;; overflow otherwise
Overflow = f(2) * ~(a2) * (b2) + ~f(2) * a(2) * ~b(2)

• Ignore ! (C language semantics)
  – What about Java? (try/catch?)
• Flag – condition code
• Sticky flag – e.g. for floating point
  – Otherwise gets in the way of fast hardware
• Trap – possibly maskable
  – MIPS has e.g. add that traps, addu that does not
Ch. 3 Summary

- Binary representations, signed/unsigned
- Arithmetic
  - Full adder, ripple-carry, carry lookahead
  - Carry-select, Carry-save
  - Overflow, negative
  - More (multiply/divide/FP) later
- Logical
  - Shift, and, or

Ch. 4 Processor Implementation

- Heart of 552 – key to project
  - Sequential logic design review (brief)
  - Clock methodology (FSD)
  - Datapath – 1 CPI
    - Single instruction, 2’s complement, unsigned
    - Control
    - Multiple cycle implementation (information only)
    - Microprogramming (information only)
    - Exceptions

Clocking Methodology

- Motivation
  - Design data and control without considering clock
  - Use Fully Synchronous Design (FSD)
    - Just a convention to simplify design process
    - Restricts design freedom
    - Eliminates complexity, can guarantee timing correctness
    - Not really feasible in real designs
    - Even in 554 you will violate FSD

Our Methodology

- Only flip-flops
- All on the same edge (e.g. falling)
- All with same clock
  - No need to draw clock signals
- All logic finishes in one cycle

Delayed Clocks (Gating)

- Problem:
  - Some flip-flops receive gated clock late
  - Data signal may violate setup & hold req’t
FSD Clocking Rules

- $T_{\text{clock}} =$ cycle time
- $T_{\text{setup}} =$ FF setup time requirement
- $T_{\text{hold}} =$ FF hold time requirement
- $T_{\text{FF}} =$ FF combinational delay
- $T_{\text{comb}} =$ Combinational delay
- FSD Rules:
  - $T_{\text{clock}} > T_{\text{FF}} + T_{\text{comb}} + T_{\text{setup}}$
  - $T_{\text{FF}} + T_{\text{comb}} > T_{\text{hold}}$

All Together

Register File?

Control Signals w/Jumps

Multi-cycle Implementation

- Clock cycle = $\max(i\text{-mem}, \text{reg-read+reg-write}, \text{ALU}, d\text{-mem})$
- Reuse combination logic on different cycles
  - One memory
    - One ALU without other adders
  - But
    - Control is more complex (later)
    - Need new registers to save values (e.g. IR)
      - Used again on later cycles
      - Logic that computes signals is reused

Multi-cycle Ctrl Signals
### Multi-cycle Steps

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Sample Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Fetch</td>
<td>IR=MEM[PC]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC=PC+4</td>
</tr>
<tr>
<td>ID</td>
<td>Decode</td>
<td>A=RF(IR[25:21])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>B=RF(IR[20:16])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Target=PC+SE(IR[15:0] &lt;&lt; 2)</td>
</tr>
<tr>
<td>EX</td>
<td>Execute</td>
<td>ALUout = A + SE(IR[15:0]) # lw/sw</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALUout = A op B # rrr</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (A==B) PC = target # beq</td>
</tr>
<tr>
<td>Mem</td>
<td>Memory</td>
<td>MEM[ALUout] = B # sw</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MDR = MEM[ALUout] #lw</td>
</tr>
<tr>
<td>WB</td>
<td>Writeback</td>
<td>Reg(IR[20:16]) = MDR # lw</td>
</tr>
</tbody>
</table>

### Multi-cycle Example (lw)

- **Start**
  - ALUsrcA = 1
  - ALUsrcB = 0
  - ALUop = 01
  - PCWriteCond
    - PCSource = 01
    - ALUSrcA = 1
    - ALUSrcB = 00
    - ALUOp = 10
- **MemRead**
  - ALUSrcA = 0
  - ALUSrcB = 11
  - ALUOp = 00
- **Push**
  - ALUSrcA = 1
  - ALUSrcB = 10
  - ALUOp = 00
- **Write MDR**
  - ALUSrcA = 0
  - ALUSrcB = 1
  - ALUOp = 00

### Microprogramming

- Alternative way of specifying control
- FSM
  - State – bubble
  - Control signals in bubble
  - Next state given by signals on arc
  - Not a great language for specifying complex events
- Instead, treat as a programming problem

### Exceptions: Big Picture

- Two types:
  - Interrupt (asynchronous) or
  - Trap (synchronous)
- Hardware handles initial reaction
- Then invokes a software exception handler
  - By convention, at e.g. 0xC00
  - O/S kernel provides code at the handler address
Exceptions: Hardware
- Sets state that identifies cause of exception
  - MIPS: in exception_code field of Cause register
- Changes to kernel mode for dangerous work ahead
- Disables interrupts
  - MIPS: recorded in status register
- Saves current PC (MIPS: exception PC)
- Jumps to specific address (MIPS: 0x80000080)
  - Like a surprise JAL – so can’t clobber $31

Exceptions: Software
- Exception handler:
  - MIPS: .ktext at 0x80000080
- Set flag to detect incorrect entry
- Nested exception while in handler
- Save some registers
- Find exception type
  - E.g. I/O interrupt or syscall
- Jump to specific exception handler

Exceptions: Software, cont’d
- Handle specific exception
- Jump to clean-up to resume user program
- Restore registers
- Reset flag that detects incorrect entry
- Atomically
  - Restore previous mode
  - Enable interrupts
  - Jump back to program (using EPC)

Implementing Exceptions
- We worry only about hardware, not s/w
- IntCause
  - 0 undefined instruction
  - 1 arithmetic overflow
- Changes to the datapath
- New states in control FSM

Review
<table>
<thead>
<tr>
<th>Type</th>
<th>Control</th>
<th>Datapath</th>
<th>Time (CPI, cycle time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-cycle</td>
<td>Comb + end update</td>
<td>No reuse</td>
<td>1 cycle, (imem + reg + ALU + dmem)</td>
</tr>
<tr>
<td>Multi-cycle</td>
<td>Comb + FSM update</td>
<td>Reuse</td>
<td>[3,5] cycles, Max(imem, reg, ALU, dmem)</td>
</tr>
<tr>
<td>We want?</td>
<td>?</td>
<td>?</td>
<td>~1 cycle, Max(imem, reg, ALU, dmem)</td>
</tr>
</tbody>
</table>
- We will use pipelining to achieve last row
Pipelining (4.5-4.9)

- Summary
  - Big Picture
  - Datapath
  - Control
  - Data Hazards
    - Stalls
    - Forwarding
  - Control Hazards
  - Exceptions

Ideal Pipelining

- Bandwidth increases linearly with pipeline depth
- Latency increases by latch delays

Ideal Pipelining

<table>
<thead>
<tr>
<th>Cycle: Instr.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+1</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+2</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+3</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i+4</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipelining Idealisms

- Uniform subcomputations
  - Can pipeline into stages with equal delay
- Identical computations
  - Can fill pipeline with identical work
- Independent computations
  - No relationships between work units
- Are these practical?
  - No, but can get close enough to get significant speedup

Complications

- Datapath
  - Five (or more) instructions in flight
- Control
  - Must correspond to multiple instructions
- Instructions may have
  - data and control flow dependences
  - I.e. units of work are not independent
    - One may have to stall and wait for another

Program Data Dependences

- True dependence (RAW)
  - \( D(i) \cap R(j) \neq \phi \)
  - \( j \) cannot execute until \( i \) produces its result
- Anti-dependence (WAR)
  - \( R(i) \cap D(j) \neq \phi \)
  - \( j \) cannot write its result until \( i \) has read its sources
- Output dependence (WAW)
  - \( D(i) \cap D(j) \neq \phi \)
  - \( j \) cannot write its result until \( i \) has written its result
Control Dependences
- Conditional branches
  - Branch must execute to determine which instruction to fetch next
  - Instructions following a conditional branch are control dependent on the branch instruction

Resolution of Pipeline Hazards
- Pipeline hazards
  - Potential violations of program dependences
  - Must ensure program dependences are not violated
- Hazard resolution
  - Static: compiler/programmer guarantees correctness
  - Dynamic: hardware performs checks at runtime
- Pipeline interlock
  - Hardware mechanism for dynamic hazard resolution
  - Must detect and enforce dependences at runtime

Pipeline Hazards
- Necessary conditions:
  - WAR: write stage earlier than read stage
    - Is this possible in IF-RD-EX-MEM-WB?
  - WAW: write stage earlier than write stage
    - Is this possible in IF-RD-EX-MEM-WB?
  - RAW: read stage earlier than write stage
    - Is this possible in IF-RD-EX-MEM-WB?
- If conditions not met, no need to resolve
- Check for both register and memory

Pipelined Datapath

Pipelined Control
- Controlled by different instructions
- Decode instructions and pass the signals down the pipe
- Control sequencing is embedded in the pipeline
Data Hazards

- Must first detect hazards

\[
\begin{align*}
\text{ID/EX.WriteRegister} &= \text{IF/ID.ReadRegister1} \\
\text{ID/EX.WriteRegister} &= \text{IF/ID.ReadRegister2} \\
\text{EX/MEM.WriteRegister} &= \text{IF/ID.ReadRegister1} \\
\text{EX/MEM.WriteRegister} &= \text{IF/ID.ReadRegister2} \\
\text{MEM/WB.WriteRegister} &= \text{IF/ID.ReadRegister1} \\
\text{MEM/WB.WriteRegister} &= \text{IF/ID.ReadRegister2}
\end{align*}
\]

Forwarding Paths (ALU instructions)

Forwarding Paths

\[\text{ALU WRITE} \rightarrow \text{ALU READ} \rightarrow \text{MEM WRITE} \rightarrow \text{MEM READ} \rightarrow \text{WBS WRITE} \rightarrow \text{WBS READ}\]

Control Flow Hazards

- What to do?
  - Always stall
  - Easy to implement
  - Performs poorly
  - 1/6th instructions are branches, each branch takes 3 cycles
  - CPI = 1 + 3 x 1/6 = 1.5 (lower bound)

Control Flow Hazards

- Predict branch not taken
- Send sequential instructions down pipeline
- Kill instructions later if incorrect
- Must stop memory accesses and RF writes
  - Including loads (why?)
- Late flush of instructions on misprediction
  - Complex
  - Global signal (wire delay)

Exceptions

- Even worse: in one cycle
  - I/O interrupt
  - User trap to OS (EX)
  - Illegal instruction (ID)
  - Arithmetic overflow
  - Hardware error
  - Etc.
- Interrupt priorities must be supported
Review

- Big Picture
- Datapath
- Control
  - Data hazards
    - Stalls
    - Forwarding or bypassing
  - Control flow hazards
    - Branch prediction
- Exceptions

IBM RISC Experience
[Agerwala and Cocke 1987]

- Internal IBM study: Limits of a scalar pipeline?
- Memory Bandwidth
  - Fetch 1 instr/cycle from I-cache
  - 40% of instructions are load/store (D-cache)
- Code characteristics (dynamic)
  - Loads – 25%
  - Stores 15%
  - ALU/RR – 40%
  - Branches – 20%
  - 1/3 unconditional (always taken)
  - 1/3 conditional taken, 1/3 conditional not taken

Simplify Branches

- Assume 90% can be PC-relative
  - No register indirect, no register access
  - Separate adder (like MIPS R3000)
  - Branch penalty reduced
- Total CPI: 1 + 0.063 + 0.085 = 1.15 CPI = 0.87 IPC

Processor Performance

\[
\text{Processor Performance} = \frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]

- In the 1980’s (decade of pipelining):
  - CPI: 5.0 => 1.15
- In the 1990’s (decade of superscalar):
  - CPI: 1.15 => 0.5 (best case)

Revisit Amdahl’s Law

- Sequential bottleneck
- Even if v is infinite
  - Performance limited by nonvectorizable portion (1-f)

\[
\lim_{v \to \infty} \frac{1}{1-f + \frac{f}{v}} = \frac{1}{1-f}
\]

Pipelined Performance Model

- g = fraction of time pipeline is filled
- 1-g = fraction of time pipeline is not filled (stalled)
Pipelined Performance Model

- \( g \) = fraction of time pipeline is filled
- \( 1-g \) = fraction of time pipeline is not filled (stalled)

Motivation for Superscalar

- Tyranny of Amdahl’s Law [Bob Colwell]
  - When \( g \) is even slightly below 100%, a big performance hit will result
  - Stalled cycles are the key adversary and must be minimized as much as possible

Superscalar Proposal

- Moderate tyranny of Amdahl’s Law
  - Ease sequential bottleneck
  - More generally applicable
  - Robust (less sensitive to \( f \))
  - Revised Amdahl’s Law:
    \[
    \text{Speedup} = \frac{1}{(1 - f) + \frac{f}{s}}
    \]

Limits on Instruction Level Parallelism (ILP)

<table>
<thead>
<tr>
<th>Authors and Year</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weiss and Smith [1984]</td>
<td>1.58</td>
</tr>
<tr>
<td>Sohi and Vajapeyam [1987]</td>
<td>1.81</td>
</tr>
<tr>
<td>Smith and Patt [1984]</td>
<td>1.96</td>
</tr>
<tr>
<td>Smith et al. [1983]</td>
<td>1.86 (Flynn’s bottleneck)</td>
</tr>
<tr>
<td>Tjaden and Flynn [1973]</td>
<td>1.50</td>
</tr>
<tr>
<td>Tjaden [1970]</td>
<td>1.96</td>
</tr>
<tr>
<td>Uht [1986]</td>
<td>2.00</td>
</tr>
<tr>
<td>Smith et al. [1989]</td>
<td>2.00</td>
</tr>
<tr>
<td>Jouppi and Wall [1988]</td>
<td>2.40</td>
</tr>
<tr>
<td>Johnson [1991]</td>
<td>2.50</td>
</tr>
<tr>
<td>Acosta et al. [1986]</td>
<td>2.79</td>
</tr>
<tr>
<td>Wedig [1982]</td>
<td>3.00</td>
</tr>
<tr>
<td>Butler et al. [1991]</td>
<td>5.8</td>
</tr>
<tr>
<td>Melvin and Patt [1991]</td>
<td>6.0</td>
</tr>
<tr>
<td>Wall [1991]</td>
<td>7.0</td>
</tr>
<tr>
<td>Kuck et al. [1972]</td>
<td>8.0</td>
</tr>
<tr>
<td>Riseman and Foster [1972]</td>
<td>51 (no control dependences)</td>
</tr>
<tr>
<td>Nicolau and Fisher [1984]</td>
<td>90 (Fisher’s optimism)</td>
</tr>
</tbody>
</table>
Classifying ILP Machines
[Jouppi, DECWRL 1991]
- Scalar pipelined
- Superpipelined
- Superscalar
- VLIW
- Superpipelined superscalar

Review Summary
- Ch. 1: Intro & performance
- Ch. 2: Instruction Sets
- Ch. 3: Arithmetic I
- Ch. 4: Data path, control, pipelining
- Details
  - Fri. 10/29 2:25-3:30 (1 hour) in EH2317
  - Closed books/notes/homeworks
  - One page handwritten cheatsheet for quick reference
  - A mix of short answer, design, analysis problems