The Execution engine
Agenda

- Engine main concept
- CISC vs. RISC
- Instruction Level Parallelism
- Pipeline
- Pipeline hazards
- uArchitecture main structures
The Generic Processor

- **Instruction supply**
  - Instruction cache
  - Branch prediction
  - Instruction decoder
  - ...

- **Execution engine**
  - Instruction scheduler
  - Register files
  - Execution units
  - ...

- **Data supply**
  - Data cache
  - TLB’s
  - ...

- **Goal** - Maximum throughput – balanced design
“New” vs. “Old” Architecture/ISA
RISC vs. CISC debate
RISC – Reduced Instruction Set Computer
CISC – Complex Instruction Set Computer

**RISC:**
- Limited number of simple instructions
- Fixed length (32) instructions + fixed field encoding
- Register to register operations
  - Load/Store architecture
- Many general purpose registers (32)
- Optional
  - 64 bit architecture
  - 3 addresses
  - Single address mode for load/store: base + displacement
  - 3-address, reg-reg arithmetic instruction
- Delayed branch

**CISC:**
- Many complex instructions
- Variable length instructions
- Memory to register operations
- Few “general/implicit” registers (8)
- Optional
  - 32/64 bit architecture
  - 2-3 addresses
  - Several address modes for load/store:
    - LD effective address
- No Delayed branch
Instruction Level Parallelism

Evolution

Basic configuration

PE=Processor Element

Instruction

Pipeline

Superscalar - In order

VLIW Static Scheduling

Superscalar - Out of Order

Fetch – Decode – Execute – Writeback
Pipelining

• **Goal:** increase throughput by using multiple units concurrently while preserving original instruction order. Can be invisible to programmer and compiler.

• **Method:** Break instruction execution into multiple steps, and execute each step in a different unit.

• **Remarks:**
  – The pipeline model is “synchronous” (equal time per step), but pipe can be stalled when necessary.
  – *Machine cycle:* the time per pipeline step (typ. 1-2 CPU cycles)
Pipelining Insights

- Pipelining doesn’t help latency of a single task (instruction), it helps throughput of entire workload (program)
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Limited by:
  - Unbalanced lengths of pipe stages
  - Clock skew
  - Inter-stage latch times
- Time to “fill” pipeline and time to “drain” it reduces speedup
Pipeline Speedup Equations

- \( \text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{avg. stall cycles} / \text{instr.} \)
- Equal clock cycle with and w/o pipelining:
  \[
  \text{Speedup} = \frac{\text{CPI}_{\text{unpipelined}}}{1 + \text{Pipeline stall cycles per instruction}}
  \]
  - special case: \( \text{CPI}_{\text{unpipelined}} = \text{pipeline depth} \)

- Equal CPI (pipelining reduces clock cycle):
  \[
  \text{Speedup} = \frac{1}{1 + \text{Pipeline stall cyc. per instr.}} \times \frac{\text{Clk cyc. unpipelined}}{\text{Clk. cyc. pipelined}}
  \]
  - special case: clock-cycle ratio = pipeline depth

- Speedup at most equals pipeline depth
- \textbf{CPI}: Avg. No. of clk cyc. between consecutive instr. completions;
  \textbf{NOT} No. of cycles required for individual instr.
Pipelining

IPC = Instructions Per Cycle
CPI = Cycles Per Instruction

- Break the work to smaller pieces

Number of stages varies
- Small: 4-5 (Pentium),
- “Superpipeline” ~14 (Pentium Pro),
- “ultra-pipeline” ~25 (Pentium 4)

Caution: we define CPI as the mean number of clock cycles between instruction completions (not the per-instruction latency)
Pipelined Datapath
Example: 5 stage pipelined Datapath
Pipeline Implementation: Comments

• Instruction, data and state travel with instruction through the pipe (pipeline registers), because they are needed in more than one stage and would otherwise be overwritten by another instruction.

• Additional minimum resource requirements:
  – Must change PC every clock cycle => additional adder
  – Fetch an instruction every clock cycle
  – Data word every clock cycle
  – Separate Mem Data Reg. for load and store
  – Read two regs and write one every clock cycle
  – Additional latches, Mux’s etc.
Pipeline Hazards

• **Limits to pipelining:** Hazards prevent next instruction from executing during its designated clock cycle
  – **Structural hazards:** Occur when HW resources cannot satisfy pipeline requirements
  – **Data hazards:** Occur when data value of operands may be incorrectly read or written as a result of the pipeline structure.
  – **Control hazards:** Pipelining of branches & other instructions that change the PC

• **Common solution** is to stall the pipeline until the hazard is resolved, inserting one or more “bubbles” in the pipeline. “Later” instructions are delayed; “earlier” ones are allowed to complete.
FIGURE 3.6 A machine with only one memory port will generate a conflict whenever a memory reference occurs.
FIGURE 3.7 The structural hazard causes pipeline bubbles to be inserted.
Example: Dual-port vs. Single-port

- Machine A: Dual ported memory
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

\[
\begin{align*}
\text{SpeedUp}_A &= \frac{\text{Pipeline Depth}}{(1 + 0)} \times \left(\frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{pipe}}}\right) \\
&= \text{Pipeline Depth} \\
\text{SpeedUp}_B &= \frac{\text{Pipeline Depth}}{(1 + 0.4 \times 1)} \times \left(\frac{\text{clock}_{\text{unpipe}}}{\text{clock}_{\text{unpipe}} / 1.05}\right) \\
&= \left(\frac{\text{Pipeline Depth}}{1.4}\right) \times 1.05 \\
&= 0.75 \times \text{Pipeline Depth}
\end{align*}
\]

\[
\frac{\text{SpeedUp}_A}{\text{SpeedUp}_B} = \frac{\text{Pipeline Depth}}{(0.75 \times \text{Pipeline Depth})} = 1.33
\]

- Machine A is 1.33 times faster
Data Hazard: Read After Write (RAW)

Instr$_i$ followed by Instr$_j$

- Instr$_j$ tries to read operand before Instr$_i$ writes it
- This is the most “intuitive” data hazard
FIGURE 3.9  The use of the result of the ADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.
Data Hazard: Write After Read (WAR)

Instrᵢ followed by Instrⱼ

- Instrⱼ tries to write operand before Instrᵢ reads it
- Can’t happen in DLX 5-stage pipeline because:
  - All instructions take 5 stages,
  - Reads are always in stage 2, and
  - Writes are always in stage 5
Data Hazard: Write After Write (WAW)

Instr\textsubscript{i} followed by Instr\textsubscript{j}

- Instr\textsubscript{j} tries to write operand before Instr\textsubscript{i} writes it
  - Leaves wrong result (Instr\textsubscript{i} not Instr\textsubscript{j})

- Can’t happen in DLX 5-stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5

- Will see WAR and WAW in later more complicated pipes
Forwarding to Avoid Data Hazard

 FIGURE 3.10 A set of instructions that depend on the ADD result use forwarding paths to avoid the data hazard.
HW Change for Forwarding

FIGURE 3.20 Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs.
The Rules of Forwarding

• Producer:
  – P1: Result exists (end of EX or MEM)
  – P2: Result conveniently available (end or middle of WB)

• Consumer:
  – C1: Reading desired (beginning or middle of ID)
  – C2: Result absolutely required (beginning of EX or MEM)

• Rules
  – \( C1 \geq P2 \): fwd. not needed
  – \( C2 \leq P1 \): must stall
  – Otherwise: forwarding avoids stalls

• (P1, P2, C1, C2 denote the times at which...)
Split-Stage ID and WB

- **Split:**
  - Read registers in 2nd half of ID
  - Write registers in 1st half of WB

- **Benefit:** (consumer ID = producer WB) is OK
FIGURE 3.12 The load instruction can bypass its results to the AND and OR instructions, but not to the SUB, since that would mean forwarding the result in "negative time."
Data Hazard Even with Forwarding

The load interlock causes a stall to be inserted at clock cycle 4, delaying the `sub` instruction and those that follow by one cycle.

**Instruction Execution**

- `lw r1, 0(r2)`
- `sub r4, r1, r6`
- `and r6, r1, r7`
- `or r8, r1, r9`
Pipeline Control Issues

• Checking for need to stall: during ID
  – compare with destination reg. ID of instructions further down the pipe (found in IR of each instr. in pipeline registers)
  – If necessary, insert NOPs into ID/EX pipeline register for as many clocks as necessary; recirculate IF/ID pipeline register

• Checking for forwarding
  – When entering EX or mem
Software Scheduling to Avoid Load Hazards

Try producing fast code for

\[ a = b + c; \]
\[ d = e - f; \]

assuming \( a, b, c, d, e, \) and \( f \) in memory.

**Slow code:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>Rb</td>
<td>b</td>
</tr>
<tr>
<td>LW</td>
<td>Rc</td>
<td>c</td>
</tr>
<tr>
<td>ADD</td>
<td>Ra</td>
<td>Rb,Rc</td>
</tr>
<tr>
<td>SW</td>
<td>a</td>
<td>Ra</td>
</tr>
<tr>
<td>LW</td>
<td>Re</td>
<td>e</td>
</tr>
<tr>
<td>LW</td>
<td>Rf</td>
<td>f</td>
</tr>
<tr>
<td>SUB</td>
<td>Rd</td>
<td>Re,Rf</td>
</tr>
<tr>
<td>SW</td>
<td>d</td>
<td>Rd</td>
</tr>
<tr>
<td>LW</td>
<td>Re</td>
<td>e</td>
</tr>
<tr>
<td>LW</td>
<td>Rf</td>
<td>f</td>
</tr>
<tr>
<td>SW</td>
<td>a</td>
<td>Ra</td>
</tr>
<tr>
<td>SUB</td>
<td>Rd</td>
<td>Re,Rf</td>
</tr>
<tr>
<td>SW</td>
<td>d</td>
<td>Rd</td>
</tr>
</tbody>
</table>

**Fast code:**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>Rb</td>
<td>b</td>
</tr>
<tr>
<td>LW</td>
<td>Rc</td>
<td>c</td>
</tr>
<tr>
<td>ADD</td>
<td>Ra</td>
<td>Rb,Rc</td>
</tr>
<tr>
<td>SW</td>
<td>a</td>
<td>Ra</td>
</tr>
<tr>
<td>LW</td>
<td>Rf</td>
<td>f</td>
</tr>
<tr>
<td>SW</td>
<td>d</td>
<td>Rd</td>
</tr>
</tbody>
</table>

Remark: “slow” code can be done with fewer registers!
Compiler Avoiding Load Stalls

<table>
<thead>
<tr>
<th></th>
<th>scheduled</th>
<th>unscheduled</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>31%</td>
<td>54%</td>
</tr>
<tr>
<td>spice</td>
<td>14%</td>
<td>42%</td>
</tr>
<tr>
<td>tex</td>
<td>25%</td>
<td>65%</td>
</tr>
</tbody>
</table>

% loads stalling pipeline
Control Hazard on Branches
Three Stage Stall

Whenever a branch enters the Pipeline 2 questions need to be resolved:
1. Is the branch taken or not?
2. What is the target of the branch? (need To be calculated)

The pipeline stage where the answer to these questions is provided is termed “Branch Resolution Stage”.
Branch Stall Impact

- If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!
- Two-part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- DLX branch merely tests if register = 0 or <> 0
- DLX Solution:
  - Move Zero test to ID/RF stage
  - Additional adder to calculate new PC in ID/RF stage:
    » data is available because of fixed-field encoding + use of immediate.
    » addition is simple, so can be squeezed into same stage with register fetch.
    » this is, of course, a speculative calculation.
  - 1 clock cycle penalty for branch versus 3
Reducing Branch Penalty to One Cycle

Branch (zero) test and target calculation are moved to ID

---

!!! Possible RAW hazard for branch condition reg.

(Error in Fig. 3.22 p. 163 in 2e textbook)
Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken
   - Execute successor instructions in sequence
   - “Squash” instructions in pipeline if branch actually taken
   - Advantage of late pipeline state update
   - 47% DLX branches not taken on average
   - PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken
   - 53% DLX branches taken on average
   - But haven’t calculated branch target address in DLX
     » DLX still incurs 1 cycle branch penalty
     » Other machines: branch target known before outcome
# Four Branch Hazard Alternatives

## #4: Delayed Branch

- Define branch to take place **AFTER** a following instruction

  ```plaintext
  branch instruction
  sequential successor_1
  sequential successor_2
  .......... 
  sequential successor_n
  branch target if taken
  ```

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- DLX uses this
- MIPS R4000 (longer pipeline, 3-clock delay):
  - schedule first delay slot
  - assume “predict not taken” for the remaining two

**Mixing architecture with implementation!!!**
Delayed Branch

• Where to get instructions to fill branch delay slot?
  – Before branch instruction
  – From the target address: only valuable when branch taken
  – From fall through: only valuable when branch not taken
  – Cancelling branches allow more slots to be filled
  – Can a nullifying branch instruction be used effectively in a branch delay slot of a conventional branch? Where can it come from?

• Compiler effectiveness for single branch delay slot:
  – Fills about 60% of branch delay slots
  – About 80% of instructions executed in branch delay slots useful in computation
  – About 50% (60% x 80%) of slots usefully filled
Dynamic Branch prediction

• **Attempts to predict:**
  – Is the branch taken or not-taken?
  – If taken, then what is the target that it should jump to?

• **Various branch predictors were studied in recent years. We divide them to two major categories:** static and dynamic.

(Much more on branch prediction later)
Data Dependency

(1) load (R1), R2
(2) mov R2, R3
(3) mov a, R1
(4) mov R3, R2
(5) mov R1, R2

• True dependency (RAW) (R2: 1>2, R3: 2>4, R1: 3>5)
• False dependencies
  – Anti dependency (WAR) (R1: 1>3, R2: 2>4)
  – Output dependency (WAW) (R2: 1>4, R2: 4>5)

RAW: Read after Write
WAR: Write after Read
WAW: Write after Write
Superscalar

• Performs more in a single cycle

- Ideally, can multiply the throughput
  - But stall occurs more frequently

2 IPC = 1/2 CPI

Examples
Intel Pentium® Proc.
Alpha 21164
Super Pipeline

- Split to shorter stages - allows higher frequency

Old clk = 0 1 2 3 4 5 6 7 8 9 10 11 12
New clk = 0 1 2 3 4 5 6 7 8 9 10 11 12

1 IPC = 1 CPI
33% higher freq!

- Ideally, can multiply the throughput, but
  - Stall penalties do not scale (e.g., control flow stall, cache misses)
  - Clock setup/hold reduces net cycle time - each instruction takes relatively longer

⇒ In this real* example: 2X stages, but performance gain is <33%

* Examples:
   Intel Pentium → Pentium II/III

Uri Weiser; Computer Architecture 101: The engine
Static Scheduling: VLIW / EPIC

• Static scheduling of instructions by the compiler
  – VLIW: Very Long Instruction Word (MultiFlow, TI6X family)
  – EPIC: Explicit Parallel Instruction set Computer (IA64)

😊 Shorter pipe, wider machine, global view
=> potentially huge ILP (wider & simpler than plain superscalar!)

😢 Many nops, sensitive to varying latencies (memory accesses)
  ⇒ Low utilization
  ⇒ Huge code size
  ⇒ Highly depends on compiler

• EPIC overcomes some of these limitations:
  – Advance loads (hide memory latency)
  – Predicated execution (avoid branches)
  – Decoder templates (reduce nops)

But at increased complexity.

Examples
Intel Itanium® proc.
DSPs
Dynamic Scheduling

• Scheduling instructions at run time, by the HW
• Advantages:
  – Works on the dynamic instruction flow:
    Can schedule across procedures, modules...
  – Can see dynamic values (memory addresses)
  – Can accommodate varying latencies and cases (e.g. cache miss)
• Disadvantages
  – Can schedule within a limited window only
  – Should be fast - cannot be too smart
Out Of Order Execution

- **In Order Execution**: instructions are processed in their program order. (Limitation to potential Parallelism.)
- **OOO**: Instructions are executed based on “*data flow*” rather than program order

**Before:**  
src -> dest  
(1) load (r10), r21  
(2) mov r21, r31  
(3) load a, r11  
(4) mov r11, r22  
(5) mov r22, r23

**After:**  
(1) load (r10), r21;  
(2) mov r21, r31;  
(3) load a, r11;  
(4) mov r11, r22;  
(5) mov r22, r23;  

<wait for loads to complete>

- **Usually highly superscalar**

Examples:  
Intel Pentium® II/III/4  
Compaq Alpha 21264
Out Of Order (cont.)

• Advantages
  – Help exploit *Instruction Level Parallelism* (ILP)
  – Help cover latencies (e.g., cache miss, divide)
  – Artificially increase the Register file size (i.e. number of registers)
  – Superior/complementary to compiler scheduler
    » Dynamic instruction window
    » Make use of more registers than the Architecture Registers

• Disadvantage: complex microarchitecture
  – Complex scheduler. Involves also
    » Large instruction window
    » Speculative execution
  – Requires reordering back-end mechanism (*retirement*) for:
    » Precise interrupt resolution
    » Misprediction/speculation recovery
    » Memory ordering
Branch Prediction

- Goal - ensure enough instruction supply by correct prefetching
- In the past - prefetcher assumed *fall-through*
  - Lose on unconditional branch (e.g., call)
  - Lose on frequently taken branches (e.g., loops)
- Branch prediction
  - Predicts whether a branch is *taken/not taken*
  - Predicts the branch target address
- Misprediction cost varies (higher w/ increased pipeline length)
- Typical Branch prediction rates: ~90%-96%
  - 4%-10% misprediction,
  - 10-25 branches between mispredictions
  - 50-125 instructions between mispredictions
- Misprediction cost increased with
  - Pipeline depth
  - Machine width
  - e.g. 3 width x 10 stages = 30 inst flushed!
Value Prediction

- Stalls due to unavailability of an operand (even with forwarding) were considered unavoidable.
- Value prediction applies the same idea as branch prediction (guessing the value of an address) to the results of the instruction producing the required operand.
- This allows execution to proceed speculatively.
- Useful especially in certain loops when there is replicated hardware that can execute in parallel.
- Some of the early work was done at the Technion (EE) by Avi Mendelson and Freddy Gabbay
Appendix
DLX - the “Course Processor” (H&P-2e; 5-stage MIPS in 3e)

- Based on modern RISC machines:
  - A simple load-store instruction set
  - Design for pipelining efficiency
  - Efficiency as a compiler target

- Basic structure:
  - 32 Integer (32-bit) general-purpose registers (R0=0)
  - 32 FP (32 bit) GPRs (used in pairs for Double Precision.)
Instruction Set

- Simple data types
- Simple addressing modes
- Simple instructions
- No implicit condition codes

(See separate page for complete instr. set)
Instruction Formats

I-type instruction

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Opcode</td>
<td>rs</td>
<td>rt</td>
<td>Immediate</td>
</tr>
</tbody>
</table>

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Opcode</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
</tbody>
</table>

Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, ...
Read/write special registers and moves

J-type instruction

<table>
<thead>
<tr>
<th></th>
<th>6</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Opcode</td>
<td>Offset added to PC</td>
</tr>
</tbody>
</table>

Jump and jump and link
Trap and return from exception

H&P3e

Fig. 2.27

48
A Simple 5-Step RISC

• **Instruction Fetch (IF)**
  – IR ← Mem[PC]; NPC ← PC+4;

• **Instr. Decode / Register Fetch (ID)**
  – A← Regs[IR$_{6..10}$]; B← Regs[IR$_{11..15}$]; Imm← ((IR$_{16}$)$^{16}$##IR$_{16..31}$);
  – (Fixed-field encoding → can read reg. while still decoding)

• **Execution / Effective address**
  – Mem ref: ALUOutput ← A+Imm
  – Reg-Reg ALU instr: ALUOutput ← A op B
  – Reg. - Imm. ALU instr: ALUOutput ← A op Imm
  – Branch: ALUOutput ← NPC+Imm; Cond ← (A op 0);
  – Jump(s): similar
5-Step RISC (cont.)

- **Memory access / branch completion (MEM)**
  - (only load, store and branches active in this step)
  - Mem ref: LMD ← Mem[ALUOutput] or Mem[ALUOut] ← B
  - Branch: if(cond) PC ← ALUOut else PC ← NPC
  - (unless taken branch or jump: PC ← NPC)

- **Write-Back (WB)**
  - Reg. - Reg. ALU instruction: Regs[IR\textsubscript{16..20}] ← ALUOutput
  - Reg. - Imm. ALU instr: Regs[IR\textsubscript{11..15}] ← ALUOutput
  - Load: Regs[IR\textsubscript{11..15}] ← LMD
5 Steps of the Datapath

1. Instruction Fetch
2. Instr. Decode
3. Reg. Fetch
4. Execute
5. Addr. Calc
6. Memory Access
7. Write Back