**Schedule in a nutshell**

1. **Memory Systems** (~Appendix C in 4th Ed)
   - Caches, VM, DRAM, microbenchmarks, optimizing SW

2. **Multiprocessors**
   - TLP: coherence, memory models, synchronization

3. **Scalable Multiprocessors**
   - Scalability, implementations, programming, ...

4. **CPUs**
   - ILP: pipelines, scheduling, superscalars, VLIWs, SIMD instructions...

5. **Widening + Future** (~Chapter 1 in 4th Ed)
   - Technology impact, GPUs, Network processors, **Multicores (!!!)**

---

**Goal for this course**

- Understand **how and why** modern computer systems are designed the way they are:
  - pipelines
  - memory organization
  - virtual/physical memory ...

- Understand **how and why** multiprocessors are built
  - Cache coherence
  - Memory models
  - Synchronization...

- Understand **how and why** parallelism is created and
  - Instruction-level parallelism
  - Memory-level parallelism
  - Thread-level parallelism...

- Understand **how and why** multiprocessors of combined SIMD/MIMD type are built
  - GPU
  - Vector processing...

- Understand **how** computer systems are adopted to different usage areas
  - General-purpose processors
  - Embedded/network processors...

- Understand the physical limitation of modern computers
  - Bandwidth
  - Energy
  - Cooling...

---

**How it all started...the fossils**

- **ENIAC** J.P. Eckert and J. Mauchly, Univ. of Pennsylvania, WW2
  - Electro Numeric Integrator And Calculator, 18.000 vacuum tubes

- **EDVAC**, J. V Neumann, operational 1952
  - Electric Discrete Variable Automatic Computer (stored programs)

- **EDSAC**, M. Wilkes, Cambridge University, 1949
  - Electric Delay Storage Automatic Calculator

- **Mark-I**... H. Aiken, Harvard, WW2, Electro-mechanic

- K. Zuse, Germany, electromech. computer, special purpose, WW2

- BARK, KTH, Gösta Neovius (was at Ericsson), Electro-mechanic early 50s

- BESK, KTH, Erik Stemme (was at Chalmers) early 50s

- **SMIL**, LTH mid 50s

---

**How do you tell a good idea from a bad idea?**

The Book: The performance-centric approach

- CPI = #execution-cycles / #instructions executed (~ISA goodness, lower is better)
- CPI * cycle time \(\Rightarrow\) performance
- CPI = CPI\text{CPU} + CPI\text{Mem}

The book rarely covers other design tradeoffs

- The cost-centric approach...
- Energy/Power-centric approach...
- Verification-centric approach...
- Complexity trade-offs

---

**The Book: Quantitative methodology**

Make design decisions based on execution statistics.
Select workloads (programs representative for usage)
Instruction mix measurements: statistics of relative usage of different components in an ISA
Experimental methodologies
  - Profiling through tracing
  - ISA simulators
Two guiding stars -- the RISC approach:

Make the common case fast
- Simulate and profile anticipated execution
- Make cost-functions for features
- Optimize for overall end result (end performance)

Watch out for Amdahl's law
- Speedup = Execution time OLD / Execution time NEW
  \[ \left(1 - \text{Fraction ENHANCED}\right) + \frac{\text{Fraction ENHANCED}}{\text{Speedup ENHANCED}} \]

Instruction Set Architecture (ISA) -- the interface between software and hardware.

Tradeoffs between many options:
- functionality for OS and compiler
- wish for many addressing modes
- compact instruction representation
- format compatible with the memory system of choice
- desire to last for many generations
- bridging the semantic gap (old desire...)
- RISC: the biggest “customer” is the compiler

ISA (instruction set architectures) trends today
- CPU families built around “Instruction Set Architectures” ISA
- Many incarnations of the same ISA
- ISAs lasting longer (~10 years)
- Consolidation in the market - fewer ISAs (not for embedded...)
- 15 years ago ISAs were driven by academia
- Today ISAs technically do not matter all that much (market-driven)
- How many of you will ever design an ISA?
- How many ISAs will be designed in Sweden?

Compilers – a moving target!
The impact of compiler optimizations

- Compiler optimizations affect the number of instructions as well as the distribution of executed instructions (the instruction mix)

Memory allocation model also has a huge impact

- Stack
  - local variables in activation record
  - addressing relative to stack pointer
  - stack pointer modified on call/return
- Global data area
  - large constants
  - global static structures
- Heap
  - dynamic objects
  - often accessed through pointers
Execution in a CPU

```
"Machine Code"
"Data"
```

CPU

Operand models

Example: C := A + B

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH [A]</td>
<td>LOAD [A]</td>
</tr>
<tr>
<td>PUSH [B]</td>
<td>ADD [B]</td>
</tr>
<tr>
<td>ADD</td>
<td>STORE [C]</td>
</tr>
<tr>
<td>POP [C]</td>
<td>STORE [C], R1</td>
</tr>
</tbody>
</table>

Stack-based machine

Example: C := A + B

```
Mem:
A:12  B:14  C:10
```

```
PUSH [A]
PUSH [B]
ADD
POP [C]
```

Mem:

```
PUSH [A]
PUSH [B]
ADD
POP [C]
```

Mem:

```
PUSH [A]
PUSH [B]
ADD
POP [C]
```

Mem:

```
PUSH [A]
PUSH [B]
ADD
POP [C]
```

Mem:
Stack-based machine
Example: C := A + B

Mem:
PUSH [A]
PUSH [B]
ADD
POP [C]

Stack-based
- Implicit operands
- Compact code format (1 instr. = 1byte)
- Simple to implement
- Not optimal for speed!!!

Accumulator-based
≈ Stack-based with a depth of one
One implicit operand from the accumulator

Mem:
PUSH [A]
ADD [B]
POP [C]

Register-based machine
Example: C := A + B

Data:

Machine Code
LD R1, [A]
LD R7, [B]
ADD R2, R1, R7
ST R2, [C]

Register-based
- Commercial success:
  - CISC: X86
  - RISC: (Alpha), SPARC, (HP-PA), Power, MIPS, ARM
  - VLIW: IA64
- Explicit operands (i.e., "registers")
- Wasteful instr. format (1 instr. ≈ 4 bytes)
- Suits optimizing compilers
- Optimal for speed!!!
Properties of operand models

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Construction</th>
<th>Implementation</th>
<th>Efficiency</th>
<th>Code Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>+</td>
<td>--</td>
<td>++</td>
<td></td>
</tr>
<tr>
<td>Accumulator</td>
<td>--</td>
<td>-</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>++</td>
<td>++</td>
<td>--</td>
<td></td>
</tr>
</tbody>
</table>

General-purpose register model dominates today
Reason: general model for compilers and efficient implementation

Instruction formats

• A variable instruction format yields compact code but instruction decoding is more complex

Generic Instruction Formats in Book

I-type

```
 Opcode Rs Rd Immediate
 0  5  5  26
```

R-type

```
 Opcode Rs1 Rs2 Rd Func
 0  5  5  5
```

J-type

```
 Opcode Offset added to PC
 0  31
```

Generic instructions (Load/Store Architecture)

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>LW R1,30(R2)</td>
<td>Rg[R1] + Mem[R2][30+Regs[R2]]</td>
</tr>
<tr>
<td>Store</td>
<td>SW 30(R2),R1</td>
<td>Mem[R2][30+Regs[R2]] + Rg[R1]</td>
</tr>
<tr>
<td>ALU</td>
<td>ADD R1,R2,R3</td>
<td>Rg[R1] + Rg[R2] + Rg[R3]</td>
</tr>
<tr>
<td>Control</td>
<td>BEQZ R1,KALLE</td>
<td>if (Regs[R1]==0) PC += KALLE * 4</td>
</tr>
</tbody>
</table>

Generic ALU Instructions

- Integer arithmetic
  - [add, sub] x [signed, unsigned] x [register, immediate]
  - e.g., ADD, ADDI, ADDUI, SUB, SUBI, SUBU, SUBUI
- Logical
  - [and, or, xor] x [register, immediate]
  - e.g., AND, ANDI, OR, ORI, XOR, XORI
- Load upper half immediate load
  - It takes two instructions to load a 32 bit immediate

Generic FP Instructions

- Floating Point arithmetic
  - [add, sub, mult, div] x [double, single]
  - e.g., ADDD, ADDF, SUBD, SUBF, ...
- Compares (sets "compare bit")
  - [lt, gt, le, ge, eq, ne] x [double, immediate]
  - e.g., LTD, GEF, ...
- Convert from/to integer, Fpregs
  - CVTF2I, CVTF2D, CVTI2D, ...
Conditional Branches

Three options:

- Condition Code: Most operations have “side effects” on set of CC-bits. A branch depends on some CC-bit.

- Condition Register. A named register is used to hold the result from a compare instruction. A following branch instruction names the same register.

- Compare and Branch. The compare and the branch is performed in the same instruction.

Simple Control

- Branches if equal or if not equal
  - BEQZ, BNEZ, cmp to register,
    \[ PC := PC + 4 + \text{immediate}_{16} \]
  - BFPT, BFPF, cmp to “FP compare bit”,
    \[ PC := PC + 4 + \text{immediate}_{16} \]

- Jumps
  - J: Jump --
    \[ PC := PC + \text{immediate}_{26} \]
  - JAL: Jump And Link --
    \[ R31 := PC + 4; PC := PC + \text{immediate}_{26} \]
  - JALR: Jump And Link Register --
    \[ R31 := PC + 4; PC := PC + \text{Reg} \]
  - JR: Jump Register --
    \[ PC := PC + \text{Reg} \] (”return from JAL or JALR”)

Important Operand Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example instruction</th>
<th>Meaning</th>
<th>When used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>Add R0, R1, #5</td>
<td>Reg[R1] + Reg[R0]</td>
<td>For constants.</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R0, R1, 100(R1)</td>
<td>Reg[R1] + \text{Mem}(100 + \text{Regs}[R1])</td>
<td>Accessing local variables.</td>
</tr>
</tbody>
</table>

Size of immediates

- Immediate operands are very important for ALU and compare operations.
- 16-bit immediates seem sufficient (75%-80%).

Implementing ISAs --pipelines

Erik Hagersten
Uppsala University

EXAMPLE: pipeline implementation

Add R1, R2, R3
Load Operation:
LD R1, mem[const+R2]

Store Operation:
ST mem[const+R1], R2

EXAMPLE: Branch to R2 if R1 == 0
BEQZ R1, R2

Initially
OP: R1==0?

Cycle 1

Cycle 2
Cycle 3

PC →

- Mem

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)

Cycle 4

PC →

- Mem

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)

Cycle 5

PC →

- Mem

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)

Cycle 6

PC →

- Mem

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)

Cycle 7

PC →

- Mem

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)

Branch → Next PC

Cycle 8

PC →

- Mem

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
Example: 5-stage pipeline

Fundamental limitations
Hazards prevent instructions from executing in parallel:

- **Structural hazards:** Simultaneous use of same resource
  If unified I+D$: LW will conflict with later I-fetch

- **Data hazards:** Data dependencies between instructions
  LW R1, 100(R2) /* result avail in 2 - 100 cycles */
  ADD R5, R1, R7

- **Control hazards:** Change in program flow
  BNEQ R1, #OFFSET
  ADD R5, R2, R3

Serialization of the execution by stalling the pipeline
is one, although inefficient, way to avoid hazards

**Fundamental types of data hazards**

- **RAW (Read-After-Write)**
  Opi+1 reads A before Opi modifies A. Opi+1 reads old A!

- **WAR (Write-After-Read)**
  Opi+1 modifies A before Opi reads A.
  Opi reads new A

- **WAW (Write-After-Write)**
  Opi+1 modifies A before Opi.
  The value in A is the one written by Opi, i.e., an old A.
Hazard avoidance techniques

Static techniques (compiler): code scheduling to avoid hazards

Dynamic techniques: hardware mechanisms to eliminate or reduce impact of hazards (e.g., out-of-order stuff)

Hybrid techniques: rely on compiler as well as hardware techniques to resolve hazards (e.g., VLIW support – later)

Cycle 3

PC

IF RegC < 100 GOTO A
RegG := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)

+ 1 R X W
Reg

Mem

Fix alt1: code scheduling

PC

IF RegC < 100 GOTO A
RegG := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)

+ 1 R X W
Reg

Mem

Fix alt2: Bypass hardware

IF ID EX M WB

Forwarding (or bypassing): provides a direct path from M and WB to EX

Only helps for ALU ops. What about load operations?

DLX with bypass

Data$ DTLB L2$ Mem

Instr$ ITLB L2$ Mem

Inst$ ITLB L2$ Mem

Instructions
Avoiding control hazards

Branch delays

Branch delays

8 cycles per iteration of 4 instructions

Need longer basic blocks with independent instr.

Avoiding control hazards

Fix1: Minimizing Branch Delay Effects

PC := PC + Imm

Fix2: Static tricks

Delayed branch (schedule useful instr. in delay slot)

Predict Branch not taken (a fairly rare case)

Predict Branch taken (a fairly common case)

Static scheduling to avoid stalls

• Scheduling an instruction from before is always safe
• Scheduling from target or from the not-taken path is not always safe; must be guaranteed that speculative instr. do no harm.
Overcoming Branches: Dynamic tricks

Erik Hagersten
Uppsala University
Sweden

Predict next PC

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)

Branch → Next PC

Cycle 4

Guess the next PC here!!

Branch Target Buffer (i.e., Cache)

NextPC

Address Tag

Next Few Instruction

Mem

Branch history table

A simple branch prediction scheme

The branch-prediction buffer is indexed by bits from branch-instruction PC values

If prediction is wrong, then invert prediction

Problem: can cause two mispredictions in a row

A two-bit prediction scheme

Requires prediction to miss twice in order to change prediction =⇒ better performance

Dynamic Scheduling Of Branches
N-level history

- Not only the PC of the BR instruction matters, also how you've got there is important
- Approach:
  - Record the outcome of the last N branches in a vector of N bits
  - Include the bits in the indexing of the branch table
- Pros/Cons: Same BR instruction may have multiple entries in the branch table

\[(N,M) \text{ prediction} = N \text{ levels of } M\text{-bit prediction}\]

Tournament prediction

- Issues:
  - No one predictor suits all applications
- Approach:
  - Implement several predictors and dynamically select the most appropriate one
- Performance example SPEC98:
  - 2-bit prediction: 7% miss prediction
  - (2,2) 2-level, 2-bit: 4% miss prediction
  - Tournaments: 3% miss prediction

Branch target buffer

- Predicts branch target address in the IF stage
- Can be combined with 2-bit branch prediction

Putting it together

- BTB stores info about taken instructions
- Combined with a separate branch history table
- Instruction fetch stage highly integrated for branch optimizations

Folding branches

- BTB often contains the next few instructions at the destination address
- Unconditional branches (and some cond as well) branches execute in zero cycles
  - Execute the dest instruction instead of the branch (if there is a hit in the BTB at the IF stage)
  - "Branch folding"
Return address stack
- Popular subroutines are called from many places in the code.
- Branch prediction may be confused!!
- May hurt other predictions
- New approach:
  - Push the return address on a [small] stack at the time of the call
  - Pop addresses on return

Architectural assumptions
<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU</td>
<td>FP ALU</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU</td>
<td>SD</td>
<td>2</td>
</tr>
<tr>
<td>LD</td>
<td>FP ALU</td>
<td>1</td>
</tr>
</tbody>
</table>

Latency=number of cycles between the two adjacent instructions
Delayed branch: one cycle delay slot

Scheduling example
for (i=1; i<=1000; i=i+1)
  x[i] = x[i] + 10;
Iterations are independent => parallel execution
loop:  
  LD  F0, 0(R1) ; F0 = array element  
  ADDD F4, F0, F2 ; Add scalar constant  
  SD 0(R1), F4 ; Save result  
  SUBI R1, R1, #8 ; decrement array ptr.  
  BNEZ R1, loop ; reiterate if R1 != 0

Can we eliminate all penalties in each iteration?
How about moving SD down?

Scheduling in each loop iteration
Original loop

can we do better by scheduling across iterations?

Scheduling in each loop iteration
Original loop

Can we do even better by scheduling across iterations?

Scheduling in each loop iteration
Original loop

Statically scheduled loop

5 instructions + 4 bubbles = 9c / iteration
5 instruction + 1 bubble = 6c / iteration

Can we even better by scheduling across iterations?

Static Scheduling of Instructions
Erik Hagersten
Uppsala University
Sweden
**Unoptimized loop unrolling 4x**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F0, 0(R1)</td>
</tr>
<tr>
<td>ADDD</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>SD</td>
<td>0(R1), F4</td>
</tr>
<tr>
<td>LD</td>
<td>F5, 0(R1)</td>
</tr>
<tr>
<td>ADDD</td>
<td>F0, F0, F2</td>
</tr>
<tr>
<td>SD</td>
<td>2(R1), F8</td>
</tr>
<tr>
<td>ST</td>
<td>0(R1), F4</td>
</tr>
<tr>
<td>ADDD</td>
<td>F10, 2(R1)</td>
</tr>
<tr>
<td>SD</td>
<td>12(R1), F12</td>
</tr>
<tr>
<td>ST</td>
<td>0(R1), F4</td>
</tr>
<tr>
<td>ADDD</td>
<td>F16, 14, F2</td>
</tr>
<tr>
<td>SUBI</td>
<td>R1, R1, #8</td>
</tr>
<tr>
<td>BNEZ</td>
<td>R1, loop</td>
</tr>
<tr>
<td>SD</td>
<td>28(R1), F16</td>
</tr>
</tbody>
</table>

**24c/ 4 iterations = 6 c / iteration**

---

**Optimized scheduled unrolled loop**

**Important steps:**
- Push loads up
- Push stores down
- Note: the displacement of the last store must be changed

**Benefits of loop unrolling:**
- Provides a larger seq. instr. window (larger basic block)
- Simplifies for static and dynamic methods to extract ILP

**24c / 4 iterations = 6 c / iteration**

From 9c to 3.5c per iteration => speedup 2.6

---

**Software pipelining**

**Symbolic loop unrolling**

- The instructions in a loop are taken from different iterations in the original loop

**Example:**

```
loop:  LD F0, 0(R1)  
       ADDD F4, F0, F2  ; drop SUBI & BNEZ
       0(R1), F4  
       LD F5, 0(R1)  
       ADDD F0, F0, F2  ; drop SUBI & BNEZ
       2(R1), F8  
       ST 0(R1), F4  
       ADDD F10, 2(R1)  
       SD 12(R1), F12  
       ADDD F16, 14, F2  ; alter to 4*8
       SUBI R1, R1, #8
       BNEZ R1, loop  
       SD 28(R1), F16
```

**Benefits of symbolic loop unrolling:**
- No data dependencies within a loop iteration
- The dependence distance is 1 iterations
- WAR hazard elimination is needed (register renaming)
- 5c / iteration, but only uses 2 FP regs (instead of 8)

---

**Software pipelining 2(3)**

**Example:**

```
loop:  LD F0, 0(R1)  
       ADDD F4, F0, F2  
       0(R1), F4  
       LD F5, 0(R1)  
       ADDD F10, 2(R1)  
       12(R1), F12  
       ADDD F16, 14, F2
       SUBI R1, R1, #8
       BNEZ R1, loop  
       SD 28(R1), F16
```

Looking at three rolled-out iterations of the loop body:

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F0, 0(R1)</td>
</tr>
<tr>
<td>ADDD</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>SD</td>
<td>0(R1), F4</td>
</tr>
<tr>
<td>SUBI</td>
<td>R1, R1, #8</td>
</tr>
<tr>
<td>BNEZ</td>
<td>R1, loop</td>
</tr>
<tr>
<td>SD</td>
<td>28(R1), F16</td>
</tr>
</tbody>
</table>
```

Execute in the same loop!!

---

**Software pipelining 3(3)**

Instructions from three consecutive iterations form the loop body:

```
< prologue code >
loop:  SD 0(R1), F4  
       ADDD F4, F0, F2  
       LD F5, 16(R1)  
       SUBI R1, R1, #8
       BNEZ R1, loop  
< prologue code >
```

- No data dependencies within a loop iteration
- The dependence distance is 1 iterations
- WAR hazard elimination is needed (register renaming)
- 5c / iteration, but only uses 2 FP regs (instead of 8)

---

**Software pipelining**

- “Symbolic Loop Unrolling”
- Very tricky for complicated loops
- Less code expansion than outlining
- Register-poor if “rotating” is used
- Needed to hide large latencies (see IA-64)
Dependencies: Revisited

Two instructions must be independent in order to execute in parallel

- Three classes of dependencies that limit parallelism:
  - Data dependencies
    \[ X \leftarrow \ldots \quad \ldots\leftarrow X \ldots \]
  - Name dependencies
    \[ \ldots \leftarrow X \]
  - Control dependencies
    \[ Y \leftarrow \ldots \quad (X > 0) \]

Getting desperate for ILP

Erik Hagersten
Uppsala University
Sweden

Multiple instruction issue per clock

Goal: Extracting ILP so that CPI < 1, i.e., IPC > 1

Superscalar:
- Combine static and dynamic scheduling to issue multiple instructions per clock
- HW finds independent instructions in "sequential" code
- Predominant: (PowerPC, SPARC, Alpha, HP-PA, x86, x86-64)

Very Long Instruction Words (VLIW):
- Static scheduling used to form packages of independent instructions that can be issued together
- Relies on compiler to find independent instructions (IA-64)

Example: A Superscalar DLX

- Issue 2 instructions simultaneously: 1 FP & 1 integer
- Fetch 64-bits/clock cycle; Integer instr. on left, FP on right
- Can only issue 2nd instruction if 1st instruction issues
- Need more ports to the register file

Statically Scheduled Superscalar DLX

Can be scheduled dynamically with Tomasulo’s alg.

Issue: Difficult to find a sufficient number of instr. to issue

(5 loops in 12 cycles)
Limits to superscalar execution

- Difficulties in scheduling within the constraints on number of functional units and the ILP in the code chunk
  - Instruction decode complexity increases with the number of issued instructions
  - Data and control dependencies are in general more costly in a superscalar processor than in a single-issue processor

Techniques to enlarge the instruction window to extract more ILP are important

Simple superscalars relying on compiler instead of HW complexity → VLIW

VLIWs: Very Long Instruction Word

VLIW: Very Long Instruction Word

- Compiler is responsible for instruction scheduling

Multicycle operations in the pipeline (floating point)

(Not a SuperScalar!)

- Integer unit: Handles integer instructions, branches, and loads/stores
- Other units: May take several cycles each. Some units are pipelined (mul, add) others are not (div)

Overlapping Execution

Erik Hagersten
Uppsala University
Sweden

Parallelism between integer and FP instructions

How to avoid structural and RAW hazards:
  - Stall in ID stage when
    - The functional unit can be occupied
    - Many instructions can reach the WB stage at the same time
  - RAW hazards:
    - Normal bypassing from MEM and WB stages
    - Stall in ID stage if any of the source operands is a destination operand of an instruction in any of the FP functional units
WAR and WAW hazards for multicycle operations

WAR hazards are a non-issue because operands are read in program order (in-order)

WAW hazards are avoided by:
- stalling the SUBF until DIVF reaches the MEM stage, or
- disabling the write to register F0 for the DIVF instruction

WAW Example:
- DIVF F0,F2,F4; FP divide 24 cycles
- SUBF F0,F8,F10; FP sub 3 cycles
- SUB finishes before DIV; out-of-order completion
- Enables out-of-order execution (& out-of-order completion)

Dynamic Instruction Scheduling

Key idea: allow subsequent independent instructions to proceed
- DIVD F0,F2,F4; takes long time
- ADDD F0,F0,F8; stalls waiting for F0
- SUBD F12,F8,F13; let this instr. bypass the ADDD

Two historical schemes used in “recent” machines:
- Tomasulo in IBM 360/91 in 1967 (also in Power-2)
- Scoreboard dates back to CDC 6600 in 1963

Simple Scoreboard Pipeline (covered briefly in this course)

- Issue: Decode and check for structural hazards
- Read operands: wait until no RAW hazard, then read operands (RAW)
- All data hazards are handled by the scoreboard mechanism

Extended Scoreboard

- Issue: Instruction is issued when:
  - No structural hazard for a functional unit
  - No WAW with an instruction in execution
- Read: Instruction reads operands when they become available (RAW)
- EX: Normal execution
- Write: Instruction writes when all previous instructions have read or written this operand (WAW, WAR)

The scoreboard is updated when an instruction proceeds to a new stage

Limitations with scoreboards

The scoreboard technique is limited by:
- Number of scoreboard entries (window size)
- Number and types of functional units
- Number of ports to the register bank
- Hazards caused by name dependencies

Tomasulo’s algorithm addresses the last two limitations

A more complicated example

- DIV F0,F2,F4; delayed a long time
- ADDD F6,F0,F8
- SUBD F12,F8,F13; can be executed right away
- MULD F6,F10,F0; delayed a few cycles

WAR and WAW avoided through “register renaming”
Tomasulo’s Algorithm

- IBM 360/91 mid 60’s
- High performance without compiler support
- Extended for modern architectures
- Many implementations (PowerPC, Pentium...)

Tomasulo’s: What is going on?

1. Read Register:
   - Rename DestReg to the Res. Station location
2. Wait for all dependencies at Res. Station
3. After Execution
   a) Put result in Reorder Buffer (ROB)
   b) Broadcast result on CDB to all waiting instructions
   c) Rename DestReg to the ROB location
4. When all preceding instr. have arrived at ROB:
   - Write value to DestReg

Simple Tomasulo’s Algorithm

- Issue
- Read operands
- Execute
- Write value to DestReg
- Rename DestReg to the ROB location
- Broadcast result on CDB to all waiting instructions
- Rename DestReg to the ROB location
Tomasulo’s: What is going on?

1. Read Register:
   - Rename DestReg to the Res. Station location
2. Wait for all dependencies at Res. Station
3. After Execution
   a) Put result in Reorder Buffer (ROB)
   b) Broadcast result on CDB to all waiting instructions
   c) Rename DestReg to the ROB location
4. When all preceding instr. have arrived at ROB:
   - Write value to DestReg

Dynamic Scheduling Past Predicted Branches

Schedule speculative instructions past branches
Dynamic Scheduling Past Predicted Branches

Wrong Prediction!!!

LD
ADD
SUB
ST

LD
ADD
SUB
ST

LD
ADD
SUB
ST

LD
ADD
SUB
ST

Wrong Prediction!!!

Do not commit!

LD
ADD
SUB
ST

LD
ADD
SUB
ST

Dynamic Scheduling Past Predicted Branches

Summing up Tomasulo’s

- Out-of-order (O-O-O) execution
- In order commit
  - Allows for speculative execution (beyond branches)
  - Allows for precise exceptions
- Distributed implementation
  - Reservation stations – wait for RAW resolution
  - Reorder Buffer (ROB)
  - Common Data Bus “snoops” (CDB)
- "Register renaming" avoids WAW, WAR
- Costly to implement (complexity and power)

Dealing with Exceptions

Erik Hagersten
Uppsala University
Sweden

Exception handling in pipelines

Example: Page fault from TLB

Must restart the instruction that causes an exception (interrupt, trap, fault) "precise interrupts" (...as well as all instructions following it.)

A solution (in-order...):
1. Force a trap instruction into the pipeline
2. Turn off all writes for the faulting instruction
3. Save the PC for the faulting instruction
   - to be used in return from exception

Guaranteeing the execution order

Exceptions may be generated in another order than the instruction execution order

<table>
<thead>
<tr>
<th>Pipeline stage</th>
<th>Problem causing exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>Page fault on instruction fetch, misaligned memory access, memory protection violation</td>
</tr>
<tr>
<td>ID</td>
<td>Undefined or illegal opcode</td>
</tr>
<tr>
<td>EX</td>
<td>Arithmetic exception</td>
</tr>
<tr>
<td>MEM</td>
<td>Page fault on data access; misaligned memory access, memory protection violation</td>
</tr>
<tr>
<td>WB</td>
<td>None</td>
</tr>
</tbody>
</table>

Example sequence:
- lw (e.g., page fault in MEM)
- add (e.g., page fault in IF)

FP Exceptions

Example:
- DIVF F0,F2,F4 24 cycles
- ADDF F10,F10,F8 3 cycles
- SUBF F12,F12,F14 3 cycles

SUBF may generate a trap before DIVF has completed!!
Revisiting Exceptions:

A pipeline implements precise interrupts iff:

All instructions before the faulting instruction can complete.

All instructions after (and including) the faulting instruction must not change the system state and must be restartable.

ROB helps the implementation in O-O-O execution.

VLIW: Very Long Instruction Word

- Independent functional units with no hazard detection
- Compiler is responsible for instruction scheduling

<table>
<thead>
<tr>
<th>Mem ref 1</th>
<th>Mem ref 2</th>
<th>FP op 1</th>
<th>FP op 2</th>
<th>SW op/Branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F13(R1)</td>
<td>LD F4(-8(R1))</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>1</td>
</tr>
<tr>
<td>LD F10(-8(R1))</td>
<td>LD F4(-8(R1))</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>2</td>
</tr>
<tr>
<td>LD F16(-8(R1))</td>
<td>LD F22(-8(R1))</td>
<td>ADDD F4,F12,F2</td>
<td>ADDD F6,F14,F2</td>
<td>NOP</td>
<td>3</td>
</tr>
<tr>
<td>LD F26(-8(R1))</td>
<td>LD F14(-24(R1))</td>
<td>NOP</td>
<td>ADDD F4,F12,F2</td>
<td>ADDD F6,F14,F2</td>
<td>NOP</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>ADDD F4,F12,F2</td>
<td>ADDD F6,F14,F2</td>
<td>NOP</td>
<td>5</td>
</tr>
<tr>
<td>SD (R1),F4</td>
<td>SD (-8(R1),F8)</td>
<td>ADDD F4,F12,F2</td>
<td>NOP</td>
<td>NOP</td>
<td>6</td>
</tr>
<tr>
<td>SD (-8(R1),F12)</td>
<td>SD (24(R1),F8)</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>7</td>
</tr>
<tr>
<td>SD (-32(R1),F28)</td>
<td>SD (40(R1),F28)</td>
<td>NOP</td>
<td>NOP</td>
<td>SUBI R1,R1,#48</td>
<td>8</td>
</tr>
<tr>
<td>SD (R1),F28</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>BNEZ R1,LOOP</td>
<td>9</td>
</tr>
</tbody>
</table>

Limits to VLIW

- Difficult to exploit parallelism
  - $N$ functional units and $K$ “dependent” pipeline stages implies $N \times K$ independent instructions to avoid stalls
- Memory and register bandwidth
- Code size
  - No binary code compatibility
- But, .... simpler hardware
  - short schedule
  - high frequency

HW support for [static] speculation and improved ILP

Erik Hagersten
Uppsala University
Sweden

HW support for static speculation

- Move LD up and ST down. But, how far?
  - Normally not outside of the basic block!
- These techniques will allow larger moves and increase the effective size of a basic block
  - Removing branches: predicate execution
  - Move LD above ST: hazard detection
  - Move LD above branch: avoid false exceptions
Compiler speculation

The compiler moves instructions before a branch so that they can be executed before the branch condition is known.

Advantage: creates longer schedulable code sequences => more ILP can be exploited.

Example:

- If (A == 0) then A = B; else A = A+4;
- Speculative code

```
LW R1,0(R3)  # Move past BR, so BR will be known
BNEZ R1,L1
LW R1,0(R2)  # Speculative code continues
ADD R1,R1,4
BEQZ R1,L3
J L2
ADD R14,R14,4
```

- Non speculative code

```
LW R1,0(R3)
BNEZ R1,L1
LW R1,0(R2)
ADD R1,R1,4
BEQZ R1,L3
J L2
ADD R14,R14,4
```

What about exceptions?

Speculative instructions

- Moving a LD up, may make it speculative.
  - Moving past a branch
  - Moving past a ST (that may be to the same address)

Issues:

- Non-intrusive
- Correct exception handling (again)
- Low overhead
- Good prediction

Example: Moving LD above a branch

```
LD.s R1, 100(R2) ; "Speculative LD" to R1
.... ; set "poison bit" in R1 if exception
BRNZ R7, #200
...
LD.chk R1 ; Get exception if poison bit of R1 is set
```

Good performance if the branch is not taken.

Example: Moving LD above a ST

```
LD.a R1, 100(R2) ; "advanced LD"
....
ST R7, 50(R3) ; invalidate entry if ALAT addr match
...
LD.c R1 ; Redo LD if entry in ALAT invalid
```

Good performance if the branch is not taken.

ALAT (advanced load address table) is an associative data structure storing tuples of: <addr, dest-reg>

Conditional execution

- Removes the need for some branches.
- Conditional Instructions
  - Conditional register move
    - CMPXCHG R1, R2, R3 ; move R3 to R1 if (R3 == 0)
  - Compare-and-swap (atomics memory operations later)
    - CMOVX R1, R2, R3 ; swap R2 and R3 if (mem(R1)== R3)
  - Avoiding a branch makes the basic block larger!!!
    - More instructions for the code scheduler to play with
  - Predicate execution
    - A more generalized technique
    - Each instruction executed if the associated 1-bit predicate REG is 1.

Predicate example

```
IF R1 > R2 then
    ADD R1, R1, #1
else
    ADD R7, 100(R2)
    ADD R2, R2, #1
```

Standard Technique

```
ADD R3,R1,R2
BNEZ R3, else
ADD R7, 100(R1)
ADD R2, R2, #1
```

5 instr executed in “then path”

2 branches
Predicate example

IF R1 > R2 then
LD R7, 100(R1)
ADD R1, R1, #1
else
LD R7, 100(R2)
ADD R2, R2, #1
end

Using Predicates

Standard Technique

One instruction sets the two predicate Regs
Each instr. in the "then" guarded by P6
Each instr. in the "else" guarded by P7
One basic block
5 instr executed in "then path"
2 branches

5 instr executed in "then path"
2 branches

HW vs. SW speculation

Advantages:
- Dynamic runtime disambiguation of memory addresses
- Dynamic branch prediction is often better than static which limits the performance of SW speculation.
- HW speculation can maintain a precise exception model

Main disadvantage:
- Complex implementation and extensive need of hardware resources (conforms with technology trends)

Example: IA64 and Itanium(I)

Erik Hagersten
Uppsala University
Sweden

Little of everything

- VLIW
- Advanced loads supported by ALAT
- Load speculation supported by predication
- Dynamic branch prediction
- “All the tricks in the book”

Itanium instructions

<table>
<thead>
<tr>
<th>Type</th>
<th>Instr 1</th>
<th>Instr 2</th>
<th>Instr 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128</td>
<td>41</td>
<td></td>
</tr>
</tbody>
</table>

- Instruction bundle (128 bits)
  - (5 bits) template (identifies I types and dependencies)
  - 3 x (41 bits) instruction
- Can issue up to two bundles per cycle (6 instr)
- The “Type” specifies if the instr. are independent
- Latencies:
  - Instruction: Latency
    - I-LD: 1
    - FP-LD: 9
    - Pred branch: 0-3
    - Misspred branch: 0-9
    - I-ALU: 0
    - FP-ALU: 4

Itanium Registers

- 128 65-bit GPR (w/ poison bit)
- 128 82-bit FP REGS
- 64 1-bit predicate REGS
- A bunch of CSRs (control/status registers)
**Dynamic register window**

- Explicit Regs (seen by the instruction)
  - Physical Regs
  - Explicit Regs (seen by main)

**Dynamic register window for GPRs**

- Physical Regs
  - Explicit Regs (seen by the main)
  - Explicit Regs (Proc A)
  - Explicit Regs (Proc B)

**Calling Procedure A**

- Procedure!!! (not processes)
  - Explicit Regs (seen by main)
  - Explicit Regs (Proc A)
  - Explicit Regs (Proc B)

**Calling Procedure B**

- Automatic passing of parameters
  - Explicit Regs (seen by main)
  - Explicit Regs (Proc A)
  - Explicit Regs (Proc B)

**Register Stack Engine (RSE)**

- Saves and restores registers to memory on register spills
- Implemented in hardware
- Works in the background
- Gives the illusion of an unlimited register stack

- This is similar to SPARC and UCB’s RISC

**Register rotation: FP and GPRs**

- Used in software pipelining
- Register renaming for each iteration
- Removes the need for prologue/epilogue
- RSE (register stack engine)
What is the alternative?

- VLIW was meant to simplify HW
- Itanium I has 230 M transistors and consumes 130W?
- Will it scale with technology?
- Other alternatives:
  - Increase cache size,
  - Increase the frequency, or,
  - Run more than one thread/chip (More about this during “Future Technologies”)

x86 Architecture

Erik Hagersten
Uppsala University
Sweden

x86 Archeology

- (8080: 1974, 6.0 kTransistors, 2MHz, 8bit)
- 8086: 1978, 29 kT, 5-10MHz, 16bit (PCI)
  - (80186:1982 ? kT, 4-40MHz, integration! )
- 80286: 1982, 0.1MT, 6-25MHz, chipset (PC-AT)
- 80386: 1985, 0.3MT, 16-33MHz, 32 bits
- 80486: 1989, 1.2MT, 25-50MHz, 18&$, FPU
- Pentium: 1993, 3.1 MT, 66 MHz, superscalar
- Pentium Pro: 1997, 5.5 MT, 200 MHz, O-O-O, 3-way superscalar
- Intel Pentium4: 2001, 42 MT, 1.5 GHz, Super-pipe, L2$ on-chip
- ...

8086 registers

- AX (Accumulator)
- BX (Base)
- CX (Count)
- DX (Data)
- SP (Stack ptr)
- BP (Base ptr)
- SI (Source index)
- DI (Destination index)
- CS (Code segment)
- DS (Data segment)
- SS (Stack segment)

Complex instructions of x86

- RISC (Reduced Instruction Set Computer)
  - LD/ST with a limited set of address modes
  - ALU instructions (a minimum)
  - Many general purpose registers
  - Simplifications (e.g., read R0 returns the value 0)
  - Simpler ISA → more efficient implementations
- x86 CISC (Complex Instruction Set Computer)
  - ALU/Memory in the same instruction
  - Complicated instructions
  - Few specialized registers (actually accumulator architecture)
  - Variable instruction length
  - x86 was lagging in performance to RISC in the 90s

x86 Micro-ops

- Newer x86 pipelines implements RISC-ish μ-ops.
- Some complex x86 instructions expanded to several micro-ops at runtime.
- The translated μ-ops may be cached in a trace-cache [in their predicted order] (first: Pentium4)
- Expanded to “loop cache” in Core-2
### x86-64
- ISA extension to x86 (by AMD 2001)
- 64-bit virtual address space
- 64-bit GP registers x16
  - x86's regs extended: rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi
  - x86-64 also has: r8, r9, ..., r15 (i.e., a total of 16 regs)
  - NOTE: dynamic register renaming makes the effective number of regs higher
- SSE: 16 128-bit SSE “vector” registers
- Backwards compatible with x86

Intel adoptions: IA-32e, EM64T, Intel64
NOTE: IA-64 is Itanium

### x86 Vector instructions
- MMX: 64 bit vectors (e.g., two 32bit ops)
- SSE: 128 bit vectors (e.g., four 32 bit ops)
- AVX: 256 bit vectors (e.g., eight 32 bit ops)
  - (in Sandy Bridge, ~Q1 2011)
- MIC: “16-way vectors”. Is this 16 x 32 bits?

### Examples of vector instructions
- **SSE_MUL** D, B, A

### Branch Delay: Each Iteration takes 11c

### Data dependency fix: pipeline delays

### From the very first lecture...
It is actually a lot worse!
Modern CPUs: "superscalars" with ~4 parallel pipelines

+ Higher throughput
- More complicated architecture
- Branch delay more expensive (more instr. missed)
- Harder to find "enough" independent instr. (need 8 instr. between write and use)

Modern CPUs: ~10-20 pipeline stages

+ Shorter cycle time (higher GHz)
- Branch delay even more expensive
- Even harder to find "enough" independent instr.

Fix: Out-of-order execution: Improving ILP

The HW may execute instructions in a different order, but will make the "side-effects" of the instructions appear in order.
Assume that LD takes a long time. The ADD is dependent on the LD. Start the SUB and ST before the ADD Update R5 and M(100) after R3

Fix: Branch prediction

The HW can guess if the branch is taken or not and avoid branch stalls if the guess is correct.
Assume the guess is "Y". The HW can start to execute these instructions before the outcome of the branch is known, but cannot allow any "side-effect" to take place until the outcome is known

Fix: Scheduling Past Branches

Predicted path
All instructions along the predicted path can be executed out-of-order

Actual path!
Wrong Prediction!!!
Throw away i.e., no side effects!
It is actually a lot worse:
Modern MEM: ~200 CPU cycles

Fix: Use cache(s)

Woops, using too much power 2007
- Running at 2x the frequency \(\Rightarrow\) will use much more than 2x the power
- It is also really hard to find enough ILP

Fix: Multicore

But now we also need to find Thread-Level Parallelism (TLP)