CPU design options

Erik Hagersten
Uppsala University
Results from mid-course eval (thanks)

Lecture speed

Talking speed

Lab difficulty

Hand-in difficulty
Interesting topic?

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not very</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Moderately</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Meaningful</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Very meaningful</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ALL NICE COMMENTS OMITTED (BUT THANKS!!)
Repetition of prev lect is good, but slides should then need more details
Great course, but needs more background/prerequisite
More real experience stuff!
Slides fg color does not work for B/W. Bringing real HW is fun!
Need guidance for exam and what to study
Hard to follow the theory
Slides need more annotations (x2)
More on current research, such as what is the current MP systems (x2)
There are schedule conflicts w other courses (+ slides need more...) x2
I miss Mr Frog (x4)
Good labs, maybe need somewhat better descriptions
The lecture room is too cold
Lower standard to pass hand-in would be better
Overlap with other course. Lab hand-in vary good
I lack a compendium. Labs and assignment are important for comprehension.
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 et 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

The Book: The performance-centric approach

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

*The book rarely covers other design tradeoffs*

- The feature centric approach...
- The cost-centric approach...
- Energy-centric approach...
- Verification-centric approach...
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
- \[ \text{Speedup} = \frac{\text{Execution time}_{\text{OLD}}}{\text{Execution time}_{\text{NEW}}} \]
- \[ \left[ (1 - \text{Fraction ENHANCED}) + \frac{\text{Fraction ENHANCED}}{\text{Speedup ENHANCED}} \right] \]
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 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?
Compiler Organization

Fortran Front-end → Intermediate Representation → High-level Optimization → Global & Local Optimization → Code Generation → Code

C Front-end → Intermediate Representation → High-level Optimization → Global & Local Optimization → Code Generation → Code

C++ Front-end → Intermediate Representation → High-level Optimization → Global & Local Optimization → Code Generation → Code

Machine-independent Translation

- Procedure in-lining
- Loop transformation

Register Allocation

Common sub-expressions

Instruction selection

constant folding
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
Operand models

Example: $C := A + B$

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

Diagram:
- Stack implicit
- Memory
- Accumulator implicit
- Register explicitly
Stack-based machine

Example: $C := A + B$

Mem:

A:12
B:14
C:10

PUSH [A]
PUSH [B]
ADD
POP [C]
Stack-based machine

Example: \( C := A + B \)

Mem:

- \( A:12 \)
- \( B:14 \)
- \( C:10 \)

PUSH [A]
PUSH [B]
ADD
POP [C]
**Stack-based machine**

Example: \( C := A + B \)

Mem:

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

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

```
Mem:

12
14
```
Stack-based machine

Example: $C := A + B$

Mem: 

- $A:12$
- $B:14$
- $C:10$

- PUSH [A]
- PUSH [B]
- ADD
- POP [C]
Stack-based machine

Example: $C := A + B$

Mem:

A: 12
B: 14
C: 10

PUSH [A]
PUSH [B]
ADD
POP [C]
Stack-based machine

Example: $C := A + B$

Mem:

A: 12
B: 14
C: 26

PUSH [A]
PUSH [B]
ADD
POP [C]
Stack-based

- Implicit operands
- Compact code format (1 instr. = 1 byte)
- 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:

- \( A:12 \)
- \( B:14 \)
- \( C:26 \)

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

| Operation & no. of operands | Address specifier 1 | Address field 1 | ... | Address specifier n | Address field n |
|-----------------------------|---------------------|-----------------|      |---------------------|-----------------|
| (a) Variable (e.g., VAX)    |                     |                 |      |                     |                 |

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>Address field 2</th>
<th>Address field 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b) Fixed (e.g., DLX, MIPS, Power PC, Precision Architecture, SPARC)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier 1</th>
<th>Address specifier 2</th>
<th>Address field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier 1</th>
<th>Address field 1</th>
<th>Address field 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c) Hybrid (e.g., IBM 360/70, Intel 80x86)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Generic Instruction Formats

### I-type

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs</th>
<th>Rd</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

### R-type

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs1</th>
<th>Rs2</th>
<th>Rd</th>
<th>Func</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>11</td>
</tr>
</tbody>
</table>

### J-type

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

---

**Dept of Information Technology | www.it.uu.se**

29
## 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>Regs[R1] ← Mem[30+Regs[R2]]</td>
</tr>
<tr>
<td>Store</td>
<td>SW 30(R2),R1</td>
<td>Mem[30+Regs[R2]] ← Regs[R1]</td>
</tr>
<tr>
<td>ALU</td>
<td>ADD R1,R2,R3</td>
<td>Regs[R1] ← Regs[R2] + Regs[R3]</td>
</tr>
</tbody>
</table>
| Control          | BEQZ R1,KALLE| if (Reg[R1]==0)  
                    |     PC ← KALLE + 4                           |
Generic ALU Instructions

- Integer arithmetic
  - [add, sub] x [signed, unsigned] x [register, immediate]
  - e.g., ADD, ADDI, ADDU, 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, SUBD, ...
- 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, ...
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 --
    \[ R31 := PC + 4; \quad PC := PC + \text{immediate}_{26} \]
  - JALR: Jump And Link Register --
    \[ R31 := PC + 4; \quad PC := PC + \text{Reg} \]
  - JR: Jump Register --
    \[ PC := PC + \text{Reg} \text{ ("return from JAL or JALR")} \]
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.
# Important Operand Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example instruction</th>
<th>Meaning</th>
<th>When used</th>
</tr>
</thead>
</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

Registers:
- Shared by all pipeline stages
- A set of general purpose registers (GPRs)
- Some specialized registers (e.g., PC)
Load Operation:

LD R1, mem[cnst+R2]
Store Operation:

ST \text{mem}[\text{cnst+R1}], R2
EXAMPLE: Branch to R2 if R1 == 0

BEQZ R1, R2

OP: R1==0?
Initially

PC

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

LD RegA, (100 + RegC)

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
Cycle 2

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
Cycle 3

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
Cycle 4

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

LD RegA, (100 + RegC)
Cycle 5

PC ➔

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
Cycle 6

PC →

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
Cycle 7

PC →

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

Branch → Next PC
Cycle 8

LD RegA, (100 + RegC)

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

![Diagram of a 5-stage pipeline with stages labeled IF, ID, EX, M, and WB.](diagram.png)
Example: 5-stage pipeline
Example: 5-stage pipeline

Instruction fetch

Instruction decode/register fetch

Execute/address calculation

Memory access

Write back

early reg write

dest data

st data

IF

ID

EX

M

WB
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

Code sequence: \( \text{Op}_i \ A \)
\( \text{Op}_{i+1} A \)

**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

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
Cycle 3

IF RegC < 100 GOTO A

"Stall"

RegC := RegC + 1

RegB := RegA + 1

"Stall"

LD RegA, (100 + RegC)

LD RegA, (100 + RegC)
Fix alt1: code scheduling

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

Swap!!
Fix alt2: Bypass hardware

- 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

IF ID EX M WB

Data$ DTLB
... L2$
... Mem

Intructions fetch
Instruction decode/ register fetch
Execute/ address calculation
Memory access

st data
dest data

IF ID EX M WB

Data$ DTLB
... L2$
... Mem

Intructions fetch
Instruction decode/ register fetch
Execute/ address calculation
Memory access

st data
dest data
Branch delays

8 cycles per iteration of 4 instructions 😞
Need longer basic blocks with independent instr.
Avoiding control hazards

Duplicate resources in ALU to compute branch condition and branch target address earlier

Branch delay cannot be completely eliminated

Branch prediction and code scheduling can reduce the branch penalty
Fix1: Minimizing Branch Delay Effects

PC := PC + Imm

(d)

IF

ID

EX

M

WB
Fix1: Minimizing Branch Delay Effects

Diagram showing the pipeline stages of instruction processing:
- **IF**: Instruction fetch
- **ID**: Instruction decode/register fetch
- **EX**: Execute/address calculation
- **M**: Memory access
- **WB**: Write back

Nodes and connections indicate flow through these stages.
Fix2: Static tricks

Delayed branch (schedule useful instr. in delay slot)
- Define branch to take place after a following instruction
- CONS: this is visible to SW, i.e., forces compatibility between generations

Predict Branch not taken (a fairly rare case)
- Execute successor instructions in sequence
- “Squash” instructions in pipeline if the branch is actually taken
- Works well if state is updated late in the pipeline
- 30%-38% of conditional branches are not taken on average

Predict Branch taken (a fairly common case)
- 62%-70% of conditional branches are taken on average
- Does not make sense for the generic arch. but may do for other pipeline organizations
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.

Goto 89
Static Scheduling of Instructions

Erik Hagersten
Uppsala University
Sweden
# 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

```
loop:     LD    F0, 0(R1)
stall
ADDD    F4, F0, F2
stall
stall
SD      0(R1), F4
SUBI    R1, R1, #8
BNEZ    R1, loop
stall
```

5 instructions + 4 bubbles = 9 cycles / iteration
(~one cycle per iteration on a vector architecture)

Can we do better by scheduling across iterations?
Scheduling in each loop iteration

Original loop:

- LD F0, 0(R1)
- ADDD F4, F0, F2
- SD 0(R1), F4
- SUBI R1, R1, #8
- BNEZ R1, loop

5 instruction + 4 bubbles = 9c / iteration

Statically scheduled loop:

- LD F0, 0(R1)
- ADDD F4, F0, F2
- SUBI R1, R1, #8
- BNEZ R1, loop
- SD 8(R1), F4

5 instruction + 1 bubble = 6c / iteration

Can we do even better by scheduling across iterations?
Unoptimized loop unrolling 4x

loop:

LD F0, 0(R1)
stall
ADDD F4, F0, F2
stall ; drop SUBI & BNEZ
stall
SD 0(R1), F4
LD F6, -8(R1)
stall
ADDD F8, F6, F2
stall ; drop SUBI & BNEZ
stall
SD -8(R1), F8
LD F10, -16(R1)
stall
ADDD F12, F10, F2
stall ; drop SUBI & BNEZ
stall
SD -16(R1), F12
LD F14, -24(R1)
stall
ADDD F16, F14, F2
SUBI R1, R1, #32 ; alter to 4*8
BNEZ R1, loop
SD -24(R1), F16

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

```
loop:           |   |   |
| LD            | F0, 0(R1) |
| LD            | F6, -8(R1) |
| LD            | F10, -16(R1) |
| LD            | F14, -24(R1) |
| ADDD          | F4, F0, F2 |
| ADDD          | F8, F6, F2 |
| ADDD          | F12, F10, F2 |
| ADDD          | F16, F14, F2 |
| SD            | 0(R1), F4 |
| SD            | -8(R1), F8 |
| SD            | -16(R1), F12 |
| SUBI          | R1, R1, #32 |
| BNEZ          | R1, loop |
| SD            | 8(R1), F16 |
```

All penalties are eliminated. CPI=1
14 cycles / 4 iterations ==> 3.5 cycles / iteration
From 9c to 3.5c per iteration ==> speedup 2.6
Software pipelining 1(3)
Symbolic loop unrolling

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

```
Iteration 0
LD
ADD
ST
SUB
```
```
BNEQ      SUB         ST          ADD          LD               BNEQ      SUB         ST          ADD          LD   Software Pipelined Loop 1
```
```
Iteration 1
```
```
BNEQ      SUB         ST          ADD          LD   BNEQ      SUB         ST          ADD          LD   Software Pipelined Loop 2
```
```
Iteration 2
```
```
BNEQ      SUB         ST          ADD          LD   BNEQ      SUB         ST          ADD          LD   Software Pipelined Loop 1
```
```
Iteration 3
```
```
BNEQ      SUB         ST          ADD          LD   BNEQ      SUB         ST          ADD          LD   Software Pipelined Loop 2
```
```
Iteration 4
```
```
Example:

**Software pipelining 2(3)**

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

- **Iteration i**:
  - `LD F0,0(R1)`
  - `ADDD F4,F0,F2`
  - `SD 0(R1),F4`
  - `SUBI R1,R1,#8`
  - `BNEZ R1,loop`

- **Iteration i+1**:
  - `LD F0,0(R1)`
  - `ADDD F4,F0,F2`
  - `SD 0(R1),F4`
  - `ADDD F4,F0,F2`
  - `SD 0(R1),F4`

- **Iteration i+2**:
  - `LD F0,0(R1)`
  - `ADDD F4,F0,F2`
  - `SD 0(R1),F4`
  - `ADDD F4,F0,F2`
  - `SD 0(R1),F4`

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        ; from iteration i
ADD F4,F0,F2           ; from iteration i+1
LD F0,-16(R1)         ; from iteration i+2
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
    
    \[ X := \ldots \]
    
    \[ \ldots := \ldots X \ldots \]
  
  • Name dependencies
    
    \[ \ldots := \ldots X \]
    
    \[ X := \ldots \]
  
  • Control dependencies
    
    \[ \text{If } (X > 0) \text{ then } \]
    
    \[ Y := \ldots \]
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)

**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)
Superscalars

Thread 1

Issue logic

2 cycles
10 cycles
30 cycles
150 cycles

Mem

Regs

2kB
64kB

2MB
1GB

1GB
2MB
64kB
2kB

SEK

£

€

Issue logic
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

<table>
<thead>
<tr>
<th>Type</th>
<th>Pipe stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int.</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Int.</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Int.</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>

- EX stage should be fully pipelined
- 1 load delay slot corresponds to three instructions!
Statically Scheduled Superscalar DLX

<table>
<thead>
<tr>
<th>Integer instruction</th>
<th>FP instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td>LD F0,0(R1)</td>
<td>1</td>
</tr>
<tr>
<td>LD</td>
<td>F6,-8(R1)</td>
<td>2</td>
</tr>
<tr>
<td>LD</td>
<td>F10,-16(R1)</td>
<td>3</td>
</tr>
<tr>
<td>LD</td>
<td>F14,-24(R1)</td>
<td>4</td>
</tr>
<tr>
<td>LD</td>
<td>F18,-32(R1)</td>
<td>5</td>
</tr>
<tr>
<td>SD</td>
<td>0(R1),F4</td>
<td>6</td>
</tr>
<tr>
<td>SD</td>
<td>-8(R1),F8</td>
<td>7</td>
</tr>
<tr>
<td>SD</td>
<td>-16(R1),F12</td>
<td>8</td>
</tr>
<tr>
<td>SD</td>
<td>-24(R1),F16</td>
<td>9</td>
</tr>
<tr>
<td>SUBI</td>
<td>R1,R1,#40</td>
<td>10</td>
</tr>
<tr>
<td>BNEZ</td>
<td>R1, LOOP</td>
<td>11</td>
</tr>
<tr>
<td>SD</td>
<td>-32(R1),F20</td>
<td>12</td>
</tr>
</tbody>
</table>

Can be scheduled dynamically with Tomasulo’s alg.

Issue: Difficult to find a sufficient number of instr. to issue
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 \(\Rightarrow\) VLIW
VLIW: Very Long Instruction Word
Very Long Instruction Word (VLIW)

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>Int op/branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
<td>LD F6,-8(R1)</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>1</td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>LD F14,-24(R1)</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>2</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>LD F22,-40(R1)</td>
<td>ADDD F4,F0,F2</td>
<td>ADDD F8,F6,F2</td>
<td>NOP</td>
<td>3</td>
</tr>
<tr>
<td>LD F26,-48(R1)</td>
<td>NOP</td>
<td>ADDD F12,F10,F2</td>
<td>ADDD F16,F14,F2</td>
<td>NOP</td>
<td>4</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>ADDD F20,F18,F2</td>
<td>ADDD F24,F22,F2</td>
<td>NOP</td>
<td>5</td>
</tr>
<tr>
<td>SD 0(R1), F4</td>
<td>SD -8(R1), F8</td>
<td>ADDD F28,F26,F2</td>
<td>NOP</td>
<td>NOP</td>
<td>6</td>
</tr>
<tr>
<td>SD -16(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),F20</td>
<td>SD -40(R1),F24</td>
<td>NOP</td>
<td>NOP</td>
<td>SUBI R1,R1,#48</td>
<td>8</td>
</tr>
<tr>
<td>SD 0(R1),F28</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>BNEZ R1,LOOP</td>
<td>9</td>
</tr>
</tbody>
</table>

VLIW will be revisited later on....
Predict next PC

PC  bubble  bubble  bubble

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!!

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

LD
ADD
SUB
ST

>=0?

LD
ADD
SUB
ST

>1?

LD
ADD
SUB
ST

>2?

Y

Y

Y
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”
Procedure calls & BTB

BTB can predict “normal” branches

Procedure A

BTB can do a good job

BTB does not stand a chance
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
Overlapping Execution

Erik Hagersten
Uppsala University
Sweden
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 (mult, add) others are not (div)
### Parallelism between integer and FP instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTD F2,F4,F6</td>
<td>IF</td>
<td>ID</td>
<td>M1</td>
<td>M2</td>
<td>M3</td>
<td>M4</td>
<td>M5</td>
<td>M6</td>
<td>M7</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>ADDD F8,F10,F12</td>
<td>IF</td>
<td>ID</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>MEM</td>
<td>WB</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>SUBI R2,R3,#8</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD F14,0(R2)</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 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
```
Dynamic Instruction Scheduling

Key idea: allow subsequent independent instructions to proceed

DIVD F0,F2,F4 ; takes long time
ADDD F10,F0,F8 ; stalls waiting for F0
SUBD F12,F8,F13 ; Let this instr. bypass the ADDD
- Enables out-of-order execution (& out-of-order completion)

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

;delayed a long time

WAR and WAW avoided through “register renaming”

Register Renaming:

DIV    F0, F2, F4
ADDD   F6, F0, F8
SUBD   F8, F10, F14 ;can be executed right away
MULD   tmp1, F10, F14 ;delayed a few cycles
Tomasulo’s Algorithm

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

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

Tomasulo’s Algorithm
Simple Tomasulo’s Algorithm

#3 DIV  F0,F2,F4
#4 ADDD  F6,F0,F8
#5 SUBD  F8,F10,F14
#6 MULD  F6,F10,F8

IF

Read operands

Issue

Mem

Int

Mem

FP

Mul1

FP

Mul2

Add

Mul

Div

Int Mem

Mem

ReOrder Buffer (ROB)

Common Data Bus (CDB)

Reg. Write Path

Res. Station

Register renaming!

ReOrder

Write Stage

Register renaming!

Reg. Write Path

0
1:
2:b
3:
4:c
5:
6:d
7:
8:e
9:

Op:div
D:F0
S1:F2
S2:F4
#3

Op:div
D:F0
S1:b
S2:c
#3

Op:div
D:F0
S1:b
S2:f
#3

Op:div
D:F0
V:c/f

V:b/c

#3

#3 DIV
D:F0
V:c/f

#3

#3 DIV
D:F0
V:b/c

#3

#3 DIV
D:F0
V:b/c

#3

111

© Erik Hagersten | user.it.uu.se/∼eh
**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

IF
Read operands
Issue
Mem
Int
Mem
FP
Add
Mul1
Mul2
Div
Res. Station
Common Data Bus (CDB)
ReOrder Buffer (ROB)
Write Stage
Reg. Write Path

#3 DIV  F0,F2,F4
#4 ADDD  F6,F0,F8
#5 SUBD  F8,F10,F14
#6 MULD  F6,F10,F8

Op
D
S1
S2
#
Op
D
S1:v/ptr
S2:v/ptr
#
Op
D:F0
S1:v
S2:v
#
D
answ
#
D
answ

Dept of Information Technology | www.it.uu.se
© Erik Hagersten | user.it.uu.se/~eh
Simple Tomasulo’s Algorithm

#3 DIV  F0,F2,F4
#4 ADDD  F6,F0,F8
#5 SUBD  F8,F10,F14
#6 MULD  F6,F10,F8

Common Data Bus (CDB)

Int Mem

Mem

FP Add

FP Mul1

FP Mul2

FP Div

ReOrder Buffer (ROB)

Write Stage

Reg. Write Path

Op

D
S1
S2
#

IF

Issue

Res. Station

0:a
1:b
2:c
3:d
4:e
5:
6:
7:
8:
9:

Common

Data Bus

ReOrder

Buffer

114

Dept of Information Technology | www.it.uu.se

© Erik Hagersten | user.it.uu.se/~eh
Simple Tomasulo’s Algorithm

IF Section:
- Read operands
- Issue

Instruction Queue (Int Mem):
- 0:a
- 1:b
- 2:c
- 3:d
- 4:e
- 5:f
- 6:g
- 7:h
- 8:i
- 9:j

Common Data Bus (CDB):
- Int Mem
- Mem

ReOrder Buffer (ROB):
- FP Add
- FP Mul1
- FP Mul2
- FP Div
- Op
  - D
  - S1
  - S2
  - #

Write Stage:
- D: answ
- #

Reg. Write Path:
- #3 DIV F0,F2,F4
- #4 ADDD F6,F0,F8
- #5 SUBD F8,F10,F14
- #6 MULD F6,F10,F8

Res. Station:
- #5

ReOrder Path:
- Op
  - D
  - S1:v/ptr
  - S2:v/ptr

#3 DIV F0,F2,F4
#4 ADDD F6,F0,F8
#5 SUBD F8,F10,F14
#6 MULD F6,F10,F8

AVDARK 2009

Dept of Information Technology | www.it.uu.se

© Erik Hagersten | user.it.uu.se/~eh
Simple Tomasulo’s Algorithm

Res. Station
Common Data Bus (CDB)
ReOrder Buffer (ROB)
Write Stage

Reg. Write Path

IF
Issue

#3 DIV F0,F2,F4
#4 ADDD F6,F0,F8
#5 SUBD F8,F10,F14
#6 MULD F6,F10,F8

0:a
1:b
2:c
3:d
4:e
5:
6:
7:
8:
9:

Op
D
S1
S2
#

Op
D
S1:v/ptr
S2:v/ptr
#

Op
D:F0
S1:v
S2:v
#

D
answ
#

D
answ

Dept of Information Technology | www.it.uu.se
© Erik Hagersten | user.it.uu.se/~eh
Simple Tomasulo’s Algorithm

#3 DIV F0,F2,F4
#4 ADDD F6,F0,F8
#5 SUBD F8,F10,F14
#6 MULD F6,F10,F8

Reg. Write Path

Res. Station

Common Data Bus (CDB)

Write Stage

ReOrder Buffer (ROB)

Op
D
S1
S2
#

Op
D
S1:v/ptr
S2:v/ptr
#

Op
D:F0
S1:v
S2:v
#

D
answ
#

© Erik Hagersten| user.it.uu.se/~eh
Simple Tomasulo’s Algorithm

Res. Station

Common Data Bus (CDB)

Reg. Write Path

IF

Issue

Mem

Int Mem

Mem

FP Add

FP Mul1

FP Mul2

FP Div

ReOrder Buffer (ROB)

Write Stage

#3 DIV F0,F2,F4
#4 ADDD F6,F0,F8
#5 SUBD F8,F10,F14
#6 MULD F6,F10,F8

Op
D
S1
S2
#

Op
D
S1:v/ptr
S2:v/ptr
#

Op
D:F0
S1:v
S2:v
#

D: answ
#

D: answ

Simple Tomasulo’s Algorithm

### IF Stage
- Read operands

### Issue Stage
- Issue operation

#### Res. Station
- **0**: b/c
- **1**: a
- **2**: b
- **3**: c
- **4**: c
- **5**: 6
- **6**: #6
- **7**: 8
- **8**: #5
- **9**: 9

#### Common Data Bus (CDB)
- Int
- Mem
- FP
- Add
- Mul1
- Mul2
- Div

#### ReOrder Buffer (ROB)
- Write Stage

#### Reg. Write Path
- #3 DIV: F0, F2, F4
- #4 ADDD: F6, F0, F8
- #5 SUBD: F8, F10, F14
- #6 MULD: F6, F10, F8

### Instruction Path
- #3: DIV F0, F2, F4
- #4: ADDD F6, F0, F8
- #5: SUBD F8, F10, F14
- #6: MULD F6, F10, F8

### Operation
- Op
  - D
  - S1
  - S2
  - #

### Data Flow
- D:
  - F0
  - ans
  - #

© Erik Hagersten | user.it.uu.se/~eh
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 preceeding instr. have arrived at ROB:
   - Write value to DestReg
Dynamic Scheduling Past Branches

Schedule speculative instructions past branches

LD
ADD
SUB
ST

LD
ADD
SUB
ST

LD
ADD
SUB
ST

LD
ADD
SUB
ST

"Predict taken"

Y

Y

Y

>=0?

>1?

<2?

"Predict taken"

"Predict taken"

"Predict taken"

Schedule speculative instructions past branches
Dynamic Scheduling Past Branches

LD ADD SUB ST

>=0?

Y

LD ADD SUB ST

>1?

Y

Wrong Prediction!!!

LD ADD SUB ST

<2?

Do not commit!

[Flowchart diagram with instructions and logic flow]
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
HW support for [static] speculation and improved ILP

Erik Hagersten
Uppsala University
Sweden
VLIW: Very Long Instruction Word

.. PC

I R B M M W
I R B M M W
I R B M M W
I R B M M W

Regs

SEK

€

£

Mem

1GB

2MB

64kB

2kB
Very Long Instruction Word (VLIW)

- 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>Int op/ branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
<td>LD F6,-8(R1)</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>1</td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>LD F14,-24(R1)</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>2</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>LD F22,-40(R1)</td>
<td>ADDD F4,F0,F2</td>
<td>ADDD F8,F6,F2</td>
<td>NOP</td>
<td>3</td>
</tr>
<tr>
<td>LD F26,-48(R1)</td>
<td>NOP</td>
<td>ADDD F12,F10,F2</td>
<td>ADDD F16,F14,F2</td>
<td>NOP</td>
<td>4</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>ADDD F20,F18,F2</td>
<td>ADDD F24,F22,F2</td>
<td>NOP</td>
<td>5</td>
</tr>
<tr>
<td>SD 0(R1), F4</td>
<td>SD -8(R1), F8</td>
<td>ADDD F28,F26,F2</td>
<td>NOP</td>
<td>NOP</td>
<td>6</td>
</tr>
<tr>
<td>SD -16(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), F20</td>
<td>SD -40(R1), F24</td>
<td>NOP</td>
<td>NOP</td>
<td>SUBI R1,R1,#48</td>
<td>8</td>
</tr>
<tr>
<td>SD 0(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

- 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 exploited.

Example:  
if (A == 0) then A = B; else A = A+4;

<table>
<thead>
<tr>
<th>Non speculative code</th>
<th>Speculative code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1,0(R3)</td>
<td>LW R1,0(R3)</td>
</tr>
<tr>
<td>BNEZ R1,L1</td>
<td>LW R14,0(R2)</td>
</tr>
<tr>
<td>LW R1,0(R2)</td>
<td>BEQZ R1,L3</td>
</tr>
<tr>
<td>J L2</td>
<td>ADD R14,R14,4</td>
</tr>
<tr>
<td>L1: ADD R1,R1,4</td>
<td>L3: SW 0(R3),R14</td>
</tr>
<tr>
<td>L2: SW 0(R3),R1</td>
<td></td>
</tr>
</tbody>
</table>

What about exceptions?

Move past BR + reg rename
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”
; create entry in the ALAT <addr,reg>

....

ST R7, 50(R3) ; invalidate entry if ALAT addr match

...

LD.c R1 ; Redo LD if entry in ALAT invalid
; remove entry in ALAT

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
    
    ```
    CMOVZ R1, R2, R3 ; move R2 to R1 if (R3 == 0)
    ```
  - Compare-and-swap (atomics memory operations later)
    
    ```
    CAS R1, R2, R3 ; swap R2 and mem(R1) 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
    LD R7, 100(R1)
    ADD R1, R1, #1
else
    LD R7, 100(R2)
    ADD R2, R2, #1
end

Standard Technique

CGT R3, R1, R2
BRNZ R3, else
LD R7, 100(R1)
ADD R1, R1, #1
BR end
else:  LD R7, 100(R2)
       ADD R2, R2, #1
end:

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

5 instr executed in “then path”
2 branches

Using Predicates

{IF R1 > R2 then P6=1;P7=0
    else P6=0;P7=1} ; //one instr!
P6: LD R7, 100(R1)
P6: ADD R1, R1, #1
P7: LD R7, 100(R2)
P7: ADD R2, R2, #1

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
- Fewer total instr
5 instr executed in “then path”
0 branch
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

- Instruction bundle (128 bits)
  - (5bits) template (identifies I types and dependencies)
  - 3 x (41bits) instruction
- Can issue up to two bundles per cycle (6 instr)
- The “Type” specifies if the instr. are independent
- Latencies:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-LD</td>
<td>1</td>
</tr>
<tr>
<td>FP-LD</td>
<td>9</td>
</tr>
<tr>
<td>Pred branch</td>
<td>0-3</td>
</tr>
<tr>
<td>Misspred branch</td>
<td>0-9</td>
</tr>
<tr>
<td>I-ALU</td>
<td>0</td>
</tr>
<tr>
<td>FP-ALU</td>
<td>4</td>
</tr>
</tbody>
</table>
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 instructions)

- Physical Regs: 127
- Explicit Regs: 63
Dynamic register window for GPRs

Explicit Regs (seen by main)

Physical Regs

Unused

Dyn. main

Global

0

31

63

Global

0

31

63

127
Calling Procedure A

Procedure!!! (....not processes)

Explicit Regs (seen by main)
Explicit Regs (seen by proc A)

Physical Regs
Unused 127
Input 85
Dyn. main 63
Global 31

Output

Input

Global
Calling Procedure B
(automatic passing of parameters)
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”)