ISA design options

Erik Hagersten
Uppsala University

ISA design options (read about it in the book!!)
- How it all started
- Instruction set overview
- The professor joke
- Options for ISA design
- Specifying HW

DARK2 in a nutshell
1. Memory Systems (caches, VM, DRAM, microbenchmarks, ...)
2. CPUs (pipelines, ILP, scheduling, Superscalars, VLIWs, embedded, ...)
3. Multiprocessors (TLP, coherence, interconnects, scalability, clusters, ...)
4. Future: (physical limitations, TLP+ILP in the CPU,...)

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
- BESK, KTH, E. Stemme (now 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 = 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
- 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
- C Front-end
- C++ Front-end

Intermediate Representation
- High-level Optimization
- Global & Local Optimization
- Code Generation

Machine-independent Translation
- Procedure in-lining
- Loop transformation
- Register Allocation
- Common sub-expressions
- Instruction selection
- Constant folding

Three different memory allocations models

- **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

Classification of ISAs

- ISAs are classified with respect to:
  - Register model
  - Number of operands for each instruction
  - Addressing modes permitted
  - Operations provided in the instruction set
  - Type and size of operands
Execution in a CPU

"Machine Code"

"Data"

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>

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

Mem:

A: 12
B: 14
C: 10

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

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
26

Example: C := A + B

Mem:

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

A:12
B:14
C:26
26

Example: C := A + B

Mem:

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

A:12
B:14
C:10
12
14

Example: C := A + B

Mem:

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

A:12
B:14
C:10
12
14

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

A:12
B:14
C:26
26

Example: C := A + B

Mem:

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

A:12
B:14
C:10
12
14

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

A:12
B:14
C:26
26
Stack-based

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

Register-based

- Commercial success:
  - X86,
  - RISCs (Alpha, SPARC, HP-PA...)
  - VLIW (IA64, ...)
- Explicit operands (i.e., "registers")
- Wasteful instr. format (1 instr. = 4 bytes)
- Suits optimizing compilers
- Optimal for speed!!

Register-based machine

Example: C := A + B

Data:

```
A: 12
B: 14
C: 26
```

```
12
14
26
```

“Machine Code”

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

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 wise
Traditional Addressing Modes (VAX)

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register deferred or indirect</td>
<td>Add R4,(R1)</td>
<td>Regs[R4] ← Regs[R4]+Mem[Regs[R1]]</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>Regs[R1] ← Regs[R1]+Mem[1001]</td>
</tr>
<tr>
<td>Memory indirect or memory deferred</td>
<td>Add R1[@(R3)]</td>
<td>Regs[R1] ← Mem[Mem[Regs[R3]]]</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>Add R1,(R2)+</td>
<td>Regs[R1] ← Regs[R1]+Mem[Regs[R2]+Regs[R2]]</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>Add R1,-(R2)</td>
<td>Regs[R1] ← Regs[R1]-Mem[Regs[R2]]</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>Regs[R1] ← Mem[100+Regs[R2]+Regs[R3]*d]</td>
</tr>
</tbody>
</table>

Are all of these addressing modes needed?

Actual use of addr. modes

- What addressing modes dominate usage?

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register deferred or indirect</td>
<td>Add R4,(R1)</td>
<td>Regs[R4] ← Regs[R4]+Mem[Regs[R1]]</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>Regs[R1] ← Regs[R1]+Mem[1001]</td>
</tr>
<tr>
<td>Memory indirect or memory deferred</td>
<td>Add R1[@(R3)]</td>
<td>Regs[R1] ← Mem[Mem[Regs[R3]]]</td>
</tr>
</tbody>
</table>

Important Addressing Modes

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register deferred or indirect</td>
<td>Add R4,(R1)</td>
<td>Regs[R4] ← Regs[R4]+Mem[Regs[R1]]</td>
</tr>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>Regs[R1] ← Regs[R1]+Mem[1001]</td>
</tr>
<tr>
<td>Memory indirect or memory deferred</td>
<td>Add R1[@(R3)]</td>
<td>Regs[R1] ← Mem[Mem[Regs[R3]]]</td>
</tr>
</tbody>
</table>

Size of displacement

- 12-16 bits cover 75% - 99% of the displacements
**Size of immediates**

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

**Operation types in the ISA**

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetical and logical</td>
<td>Integer arithmetic and logical operations: add, and, subtract, or</td>
</tr>
<tr>
<td>Data transfer</td>
<td>Loads/stores (move instructions on machines with memory addressing)</td>
</tr>
<tr>
<td>Control</td>
<td>Branch, jump, procedure call and return</td>
</tr>
<tr>
<td>System</td>
<td>Operating system call, virtual memory management instructions</td>
</tr>
<tr>
<td>Floating point</td>
<td>Floating-point operations: add, multiply, ...</td>
</tr>
<tr>
<td>Decimal</td>
<td>Decimal add, decimal multiply, decimal-to-character conversions</td>
</tr>
<tr>
<td>String</td>
<td>String move, string compare, string search</td>
</tr>
</tbody>
</table>

**Control instructions**

- Conditional branches
- Unconditional branches (jumps)

Conditional branches dominate by far
Intuition: program loops are common!

**Size of branch displacement**

- 8-bit branch displacements cover most cases
Intuition: most program loops are tight!
**Branch condition evaluation**

<table>
<thead>
<tr>
<th>Name</th>
<th>How?</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition Code (CC) are manipulated</td>
<td>Special bits are CC set free Extra state manipulated</td>
<td>Test general purpose register Uses up registers</td>
<td>Compare and branch Compare is part of branch One instr. instead of two Extra work per instr.</td>
</tr>
</tbody>
</table>

**Instruction formats**

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

**Summary: Classification of ISAs**

- ISAs are classified with respect to:
  - Register model
  - Number of operands for each instruction
  - Addressing modes permitted
  - Operations provided in the instruction set
  - Type and size of operands

**ISAs versus compilers**

Rules of thumb when designing an ISA:

- Regularity (operations, data types and addressing modes should be orthogonal)
- Provide primitives, not high-level constructs. Complex instructions are often too specialized.
The impact of compiler optimizations

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

A generic architecture

Load/store architecture (32 bits)

- Many (32) gp integer registers (GPR) and single precision floating point registers (GPR0 = 0)
- Fixed instruction width and format
- Addressing modes: immediate and displacement
- Supported data types: bytes, half word (16 bits), word (32 bits), single and double precision IEEE floating points

Generic instructions

<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>
<tr>
<td>Control</td>
<td>BEQZ R1,KALLE</td>
<td>if Regs[R1] == 0</td>
</tr>
</tbody>
</table>

Specifying hardware

- <--n : Transfer n bits
- R7n : Bit n of register R7
- R20..7 : The most significant byte of R2 (Big endian!)
- 08 : A byte of all zeroes (repeat the field n times)
- M[40] : byte 40 in memory
- 08 # # M[40] : Concatenate zero-byte (MSB) with the byte @mem(40)
Generic Move Instructions

- Load and Store
  - LB, LBU, SB -- byte chunks
  - LH, LHU, SH -- half word chunks
  - LW, SW -- word chunks
  - LF, SF -- word chunks to floating point regs
  - LD, SD double precision to FP regs (2 regs per OP)

- Register to Register moves

Examples Hardware Descriptions

Load and Store

- LB, LBU, SB -- byte chunks
- LH, LHU, SH -- half word chunks
- LW, SW -- word chunks
- LF, SF -- word chunks to floating point regs
- LD, SD double precision to FP regs (2 regs per OP)

Examples Hardware Descriptions (2)

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 instruction to load a 32 bit immediate
More Generic ALU Ops

- Shifts
  - [left, right] x [logical, arithmetic] x [immediate, reg]
  - e.g., SLL, SRAI, ...
- Set conditional
  - [lt, gt, le, ge, eq, ne] x [immediate, reg]
  - e.g., SLT, SGEI, ...
  - Puts a 1 or a 0 in the destination register

Generic Instruction Formats

- **I-type**
  - Opcode
  - Rs
  - Rd
  - Immediate
  - Format: 0 Opcode Rs Rd Immediate

- **R-type**
  - Opcode
  - Rs1
  - Rs2
  - Rd
  - Func
  - Format: 0 Opcode Rs1 Rs2 Rd Func

- **J-type**
  - Opcode
  - Offset added to PC
  - Format: 0 Opcode Offset added to PC

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 + Immediate
  - BFTP, BFPP, cmp to "FP compare bit", PC := PC + 4 + Immediate
- Jumps
  - J: Jump -- PC := PC + Immediate
  - JAL: Jump And Link -- R31 := PC + 4; PC := PC + Immediate
  - JALR: Jump And Link Register -- R31 := PC + 4; PC := PC + Reg
  - JR: Jump Register -- PC := PC + Reg ("return from JAL or JALR")
  - TRAP: go to OS via a vectored address (more later)
  - RTF: ReTurn From user code (more later)
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
Initially

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

Cycle 1

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

Cycle 2

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

Cycle 3

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

PC → Mem

I R X W
Regs

PC → Mem

I R X W
Regs

PC → Mem

I R X W
Regs

Cycle 3

+
Cycle 4

LD RegA, (100 + RegC)

IF RegC < 100 GOTO A

RegC := RegC + 1

RegB := RegA + 1

Cycle 5

IF RegC < 100 GOTO A

RegC := RegC + 1

RegB := RegA + 1

LD RegA, (100 + RegC)

Cycle 6

IF RegC < 100 GOTO A

RegC := RegC + 1

RegB := RegA + 1

LD RegA, (100 + RegC)

Cycle 7

IF RegC < 100 GOTO A

RegC := RegC + 1

RegB := RegA + 1

LD RegA, (100 + RegC)

Branch ➔ Next PC
Cycle 8

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

PC →

1  | R  | X  | W  
---|----|----|----|
    |    |    |    |
|    |    |    |    |
|    |    |    |    |

Regs

Mem

Example: 5-stage pipeline

IF  ID  EX  M  WB

(d) s1 s2

st data

PC
Example: 5-stage pipeline

Pipeline Challenges
- Balance the pipeline stages
- Handle the feed-back cases
- Minimize pipeline stalls
- Predict and perform speculative work
- Undo speculative work

Fundamental limitations
Hazards prevent instructions from executing in parallel:

- **Structural hazards**: Simultaneous use of same resource
  - If unified I+DS: 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

<table>
<thead>
<tr>
<th>Code sequence</th>
<th>Instruction (Op_i A)</th>
<th>Instruction (Op_{i+1} A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAW (Read-After-Write)</td>
<td>Op_i reads A before Op_i modifies A. Op_{i+1} reads old A!</td>
<td></td>
</tr>
<tr>
<td>WAR (Write-AFTER-Read)</td>
<td>Op_{i+1} modifies A before Op_i reads A. Op_i reads new A</td>
<td></td>
</tr>
<tr>
<td>WAW (Write-AFTER-Write)</td>
<td>Op_{i+1} modifies A before Op_i. The value in A is the one written by Op_i, i.e., an old A.</td>
<td></td>
</tr>
</tbody>
</table>
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)

Data dependency

Cycle 3

Fix: code scheduling

Swap!!
Fix: Bypass hardware

- Forwarding (or bypassing): provide a direct path from M and WB to EX
- Only helps for ALU ops. What about load operations?

A more optimized DLX pipeline...

Avoiding control hazards

Branch delays

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
Taking a Branch

```
PC := PC + Imm
```

Fix1: Minimizing Branch Delay Effects

```
IF IF ID ID EX EX M M WB WB
```

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.
Evaluating branch hazard avoidance techniques

Pipeline speedup = \( \frac{\#\text{stages}}{1 + \text{Branch frequency} \times \text{Branch penalty}} \)

<table>
<thead>
<tr>
<th>Scheduling scheme</th>
<th>Branch penalty for Integ. Pgm</th>
<th>CPI</th>
<th>Speedup vs Unpipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline</td>
<td>1</td>
<td>1.17</td>
<td>4.3</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td>1.17</td>
<td>4.3</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>0.69</td>
<td>1.12</td>
<td>4.5</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.21</td>
<td>1.04</td>
<td>4.8</td>
</tr>
</tbody>
</table>

Dynamic branch prediction

Branches limit performance because:
- Branch penalties are high
- Prevent a lot of ILP from being exploited

Solution: Dynamic branch prediction to predict the outcome of conditional branches.

Benefits:
- Reduce time to determine branch condition
- Reduce time to calculate the branch target address

FIX3: Predict next PC

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

Cycle 4

Guess the next PC here!!

IF RegC < 100 GOTO A
RegC := RegC + 1
RegB := RegA + 1
LD RegA, (100 + RegC)
More on Dynamic Branch Prediction

Erik Hagersten
Uppsala University
Sweden

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 misspredictions 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

\[(N,M)\] prediction = N levels of M-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 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 conditional 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"

Branch prediction penalties

<table>
<thead>
<tr>
<th>Instruction in buffer</th>
<th>Prediction</th>
<th>Actual branch</th>
<th>Penalty cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Taken</td>
<td>Taken</td>
<td>0</td>
</tr>
<tr>
<td>Yes</td>
<td>Taken</td>
<td>Not taken</td>
<td>2</td>
</tr>
<tr>
<td>Yes</td>
<td>Not taken</td>
<td>Not taken</td>
<td>0</td>
</tr>
<tr>
<td>Yes</td>
<td>Not taken</td>
<td>Taken</td>
<td>2</td>
</tr>
<tr>
<td>No</td>
<td>Taken</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Not taken</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Itanium: ranging from 0 to 9 cycles...

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

Summary Pipelining

Pipelining:
- Speeds up throughput, not latency
- Speedup ≤ #stages
- Hazards are fundamental limits:
  - Structural: need more HW
  - Data (RAW, WAR, WAW): need forwarding and compiler scheduling
  - Control: delayed branch one solution
Overlapping Execution

Erik Hagersten
Uppsala University
Sweden

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)

Exception handling in pipelines

Must restart an instruction that causes an exception (interrupt, trap, fault) “precise interrupts”

(...as well as all instructions following it.)

A solution:
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
   - may need to save multiple PC values

Multicycle operations in the pipeline (floating point)

- 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

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
- 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

Example of a WAR hazard:
DIVF \( F_0, F_2, F_4 \) FP divide 24 cycles
...
SUBF \( F_0, F_8, F_10 \) FP sub 3 cycles
SUB finishes before DIV; out-of-order completion

WAW hazards are avoided by:
- Stalling the SUBF until DIVF reaches the MEM stage, or
- Disabling the write to register \( F_0 \) for the DIVF instruction

FP Exceptions

Example:
DIVF \( F_0, F_2, F_4 \) 24 cycles
ADDF \( F_10, F_10, F_8 \) 3 cycles
SUBF \( F_12, F_12, F_14 \) 3 cycles

SUBF may generate a trap before DIVF has completed!!

Dynamic Instruction Scheduling

Key idea: allow subsequent independent instructions to proceed
- DIVD \( F_0, F_2, F_4 \); takes long time
- ADDD \( F_10, F_0, F_8 \); stalls waiting for \( F_0 \)
- SUBD \( F_12, F_8, F_13 \); 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

- **Issue**: Decode and check for structural hazards
- **Read operands**: wait until no data 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 complicated example

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIV F0,F2,F4</td>
<td>RAW</td>
<td>;delayed a long time</td>
</tr>
<tr>
<td>ADDD F6,F0,F8</td>
<td>WAR</td>
<td></td>
</tr>
<tr>
<td>SUBD F8,F10,F14</td>
<td>RAW</td>
<td></td>
</tr>
<tr>
<td>MULD F6,F10,F8</td>
<td></td>
<td>WAR and WAW avoided through “register renaming”</td>
</tr>
<tr>
<td>DIV F0,F2,F4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD F6,F0,F8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBD tmp1,F10,F14</td>
<td>RAW</td>
<td>;can be executed right away</td>
</tr>
<tr>
<td>MULD tmp2,F10,tmp1</td>
<td></td>
<td>;delayed a few cycles</td>
</tr>
</tbody>
</table>
Tomasulo’s Algorithm
(touched briefly in this course)

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

Simple Tomasulo’s Algorithm

<table>
<thead>
<tr>
<th>Res. Station</th>
<th>Common Data Bus (CDB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int Mem</td>
<td>Op:div</td>
</tr>
<tr>
<td>Mem</td>
<td>D:F0</td>
</tr>
<tr>
<td>IF</td>
<td></td>
</tr>
<tr>
<td>Reg. Write</td>
<td></td>
</tr>
<tr>
<td>Write Stage</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- IF: Instruction Fetch
- Mem: Memory
- Int: Integer
- FP: Floating Point
- Add: Addition
- Mul: Multiplication
- Div: Division
- ROB: ReOrder Buffer
- CDB: Common Data Bus

Diagram:
- OP: D/F0
- S1: F2
- S2: F4
- #3

Diagram:
- D: ans
- F: ans
- V: c/f

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

Diagram:
- D: ans
- F: ans
- V: c/f

Diagram:
- Path:
  - #3 DIV D:F0, S1:v, S2:v
  - #4 ADDD D:F6, S1:v/ptr, S2:v/ptr

Diagram:
- Path:
  - #3 DIV D:F0, S1:v, S2:v
  - #4 ADDD D:F6, S1:v/ptr, S2:v/ptr
Simple Tomasulo’s Algorithm

- **IF**
- **Read operands**
- **Issue**
- **Mem**
- **Int**
- **Mem**
- **FP**
- **Add**
- **FP**
- **Mul1**
- **FP**
- **Mul2**
- **FP**
- **Div**
- **Mem**

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

- **Common Data Bus (CDB)**
- **ReOrder Buffer (ROB)**
- **Write Stage**
- **Reg. Write Path**

- **#3 DIV F0,F2,F4**
- **#4 ADD F6,F0,F8**
- **#5 SUB F8,F10,F14**
- **#6 MULD F6,F10,F8**

- **Op**
- **D**
- **Op**
- **D**
- **Op**
- **D**
- **Op**
- **D**
- **Op**
- **D**

- **D**
- **Div**
- **answer**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
- **S1:v/ptr**
- **S2:v/ptr**
Simple Tomasulo’s Algorithm

**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)

Dynamic Scheduling Past Branches

**Dynamic Scheduling Past Branches**

Schedule speculative instructions past branches

Wrong Prediction!!!

Do not commit!
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.

Goto 147

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

Static Scheduling of Instructions

Erik Hagersten
Uppsala University
Sweden

Scheduling example

For \( i = 1; i \leq 1000; i = i + 1 \)

\[ x[i] = x[i] + 10; \]

Iterations are independent => parallel execution

Loop:

\begin{align*}
  \text{LD} & \quad \text{F0}, 0(R1) & \text{FO} = \text{array element} \\
  \text{ADD} & \quad \text{F4}, \text{F0}, \text{F2} & \text{Add scalar constant} \\
  \text{SD} & \quad 0(R1), \text{F4} & \text{Save result} \\
  \text{SUBI} & \quad R1, R1, #8 & \text{Decrement array ptr.} \\
  \text{BNEZ} & \quad R1, \text{loop} & \text{Reritate if R1 \neq 0} 
\end{align*}

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
      SD    0(R1), F4  stall
      SUBI  R1, R1, #8
      BNEZ  R1, loop
```

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

Can we do better by scheduling across iterations?

Can we do even better by scheduling across iterations?

5 instruction + 4 bubbles  = 9c / iteration

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

5 instruction + 1 bubble = 6c / iteration

```
loop:  LD   F0, 0(R1)   stall
      ADDD  F4, F0, F2  stall
      SD    0(R1), F4  stall
      SUBI  R1, R1, #8
      BNEZ  R1, loop
      # alter to 4*8
```

24c/ 4 iterations = 6 c / iteration

Unoptimized loop unrolling 4x

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

Vector architectures (a footnote)
CRAY, NEC, Fujitsu, ...

- 8 vector register contains 64 vector entries each
- A single LD/ST instr loads/stores entire vectors
- A single ALU instr V1 ← V2 op V3
- 64 bit mask vectors make execution conditional
- Overlaps Mem and ALU ops
- One form of “SIMD” -- Single Instruction Multiple Data
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

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

Software pipelining 2(3)

**Example:**

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

6 cycles / iteration == speedup 3.3
```

Software pipelining 3(3)

**Instructions from three consecutive iterations form the loop body:**

```assembly
loop: SD 0(R1), F4
       ADDD F4, F0, F2
       LD F0, -16(R1)
       SUBI R1, R1, #8
       BNEZ R1, loop

Execute in the same loop!!
```
Software pipelining

- "Symbolic Loop Unrolling"
- Very tricky for complicated loops
- Less code expansion than outlining
- Register-lean if "rotating" is used
- Needed to hide large latencies (see IA-64)

Detecting data dependencies

- Finding dependencies is fundamental to
  - perform instruction scheduling;
  - determine the degree of parallelism in loops; and
  - eliminate name dependencies

```c
for (i = 1; i <= 100; i = i+1) {
    A[i] = B[i] + C[i];
    D[i] = A[i] + E[i];
}
```

The absence of loop-carried dependencies increases the amount of exploitable parallelism

Loop-carried dependencies

A loop iteration is often dependent on results calculated in an earlier iteration.

*Example:*  
```c
for (i = 6; i <= 100; i = i+1) 
    Y[i] = Y[i-5] + Y[i];
```

- This loop has a dependence distance of 5 and we can extract ILP in 5 consecutive iterations
  - What dependencies can the compiler detect?

The GCD test

- A simple test to decide whether there is any dependencies between loop iterations:
  - If loop carried dependencies exist then:  
    
    \[ \text{GCD}(c,a) \text{ divides } (d - b) \]

*Example:*  
```c
for (i = 1; i <= 100; i = i + 1) 
```

We have \(a=2, b=3, c=2\) and \(d=0\); \(\text{GCD}(2,2) = 2\) which does not divide \((0-3) = -3\) =>

- There are no loop carried dependencies in this loop
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)

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</td>
</tr>
<tr>
<td>FP</td>
<td>IF</td>
</tr>
<tr>
<td>Int.</td>
<td>IF</td>
</tr>
<tr>
<td>FP</td>
<td>IF</td>
</tr>
<tr>
<td>Int.</td>
<td>IF</td>
</tr>
<tr>
<td>FP</td>
<td>IF</td>
</tr>
</tbody>
</table>
Statically Scheduled Superscalar DLX

<table>
<thead>
<tr>
<th>Integer Instruction</th>
<th>FP Instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R0, R1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LD R0, R1, R2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>LD R0, R1, R2, R3</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>LD R0, R1, R2</td>
<td>ADD R0, R0, R2</td>
<td>5</td>
</tr>
<tr>
<td>ADD R0, R0, R2</td>
<td>ADD R0, R0, R2</td>
<td>6</td>
</tr>
<tr>
<td>SUB R0, R0, R2</td>
<td>ADD R0, R0, R2</td>
<td>7</td>
</tr>
<tr>
<td>SUB R0, R0, R2</td>
<td>ADD R0, R0, R2</td>
<td>8</td>
</tr>
<tr>
<td>SUB R0, R0, R2</td>
<td>ADD R0, R0, R2</td>
<td>9</td>
</tr>
<tr>
<td>SUB R0, R0, R2</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>SUB R0, R0, R2</td>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>

Can be scheduled dynamically with Tomasulo’s alg.

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

Dependencies: Revisited

Two instructions must be independent in order to execute in parallel

Three classes of dependencies that limit parallelism:
- Data dependencies
- Name dependencies
- Control dependencies

- Dependencies are properties of the program
- Can lead to hazards which are properties of the implementation

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

Simple superscalar relying on compiler instead of HW complexity → VLIW

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

VLIW: Very Long Instruction Word
**Very Long Instruction Word (VLIW)**

- Independent functional units with no hazard detection

Compiler is responsible for instruction scheduling

```
Mem off 1   Mem off 2   FP off 1   FP off 2   Dep/Brach   Clock
LD F10,36(R1)   LD F14,24(R1)   1
LD F18,32(R1)   LD F22,40(R1)   ADD F4,F0,F2   ADD F8,F6,F2   2
LD F26,48(R1)   ADD F12,F10,F2   ADD F16,F14,F2   3
ADD F28,F26,F2   ADD F24,F22,F2   4
ADD F30,F34,F32   ADD F38,F36,F32   5
SD R11,F4   SD R11,F8   ADD F38,F36,F32   6
SD -16(R1),F12   SD -24(R1),F8   7
SD -32(R1),F20   SD -40(R1),F24   SUB R11,R1,R1   8
SD R11,F28   BNEZ R1,LOOP   9
```

**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
  - Complexity increases with number of functional units
  - Code size

- No binary code compatibility

But, .... simpler hardware
- short schedule
- high frequency

**HW support for speculation**

Speculative execution = execute instructions before all control dependencies are resolved

A combination of three main ideas we’ve covered
- Dynamic Instruction scheduling; take advantage of ILP
- Dynamic branch prediction; allows instruction scheduling across branches
- Hardware based speculation uses a data-flow approach: instructions execute when their operands are available

Erik Hagersten
Uppsala University
Sweden
HW support for static speculation

- Move LD up and ST down. But, how far?
- These techniques will allow larger moves and increase the effective size of a basic block
  - Regrouping instructions: trace scheduling, superblocks
  - Removing branches: predicate execution
  - Move LD above ST: hazard detection
  - Move LD above branch

Trace scheduling 1(2)

Creates a sequence of instructions that are likely to be executed -- a trace.

Two steps:
- Trace selection: Find a likely sequence of basic blocks (trace) across statically predicted branches (e.g. if-then-else)
- Trace compaction: Schedule the trace to be as efficient as possible while preserving correctness in the case the prediction is wrong

- Yields more instruction level parallelism
- Efficient static branch prediction key to success

Trace scheduling 2(2)

- The leftmost sequence is chosen as the most likely trace
- The assignment to B is control dependent on the if statement.
- Trace compaction has to respect data dependencies
- The rightmost (less likely) trace has to be augmented with fix up code

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;
```

Non speculative code

<table>
<thead>
<tr>
<th>Non speculative code</th>
<th>Speculative code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1(R3)</td>
<td>LW R1(R3)</td>
</tr>
<tr>
<td>BNEZ R1,L1</td>
<td>LW R1(R3)</td>
</tr>
<tr>
<td>LW R1(R2)</td>
<td>BEQZ R1,L3</td>
</tr>
<tr>
<td>J R2</td>
<td>ADD R14,R14</td>
</tr>
<tr>
<td>L1: ADD R1,R14</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?
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 load
;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 covered later)

- Predicate execution
  - A more generalized technique
  - Each instruction executed if the associated 1-bit predicate REG is 1.
**Predicate example**

**Standard Technique**

```plaintext
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

**Using Predicates**

```plaintext
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

**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)

**Summary**

- **Software (compiler) tricks:**
  - Loop unrolling
  - Software pipelining
  - Static instruction scheduling (with register renaming)
  - Trace scheduling (implies static branch prediction)
  - Speculative execution

- **Hardware tricks:**
  - Dynamic instruction scheduling
  - Dynamic branch prediction
  - Multiple issue
    - Superscalar
    - VLIW
  - Conditional instructions
  - Speculative execution
Example: IA64 and Itanium

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)
  - (5 bits) template (identifies I types and dependencies)
  - 3 x (41 bits) instruction
- Can issue up to two bundles per cycle (6 instr)
- 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>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
- 8 64-bit branch registers
- A bunch of CSRs (control/status registers)
Dynamic register window

Explicit Regs (seen by the instructions)

Physical Regs

64

0

127

165

Dynamic register window for GPRs

Explicit Regs (seen by main)

Physical Regs

63

0

31

31

0

166

Calling procedure A

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

Physical Regs

85

10

63

31

0

167

Calling Procedure B (automatic passing of parameters)

Explicit Regs (seen by main)

Explicit Regs (Proc A)

Explicit Regs (Proc B)

Physical Regs

85

10

63

31

4

4

168
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

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
- Is this the best you can do with 230 M transistors and 130W?
- Will it scale with technology?
- Other alternatives:
  - Increase cache size,
  - Increase the frequency, or,
  - Run more than one thread/chip

Technology Questions for the Future:
Where is technology pushing us?
Can we continue this trend?

- Huge monolithic single-thread monster CPUs
- Diminishing return on:
  - pipeline stages
  - parallel pipelines
- The memory wall is the limit
- should we continue share memory?
- Will we need parallel programs?
- What is in the technology crystal ball?

Technology "Improvements"

Quantitative data and trends according to V. Agarwal et al., ISCA 2000
Based on SIA (Semiconductor Industry Association) prediction, 1999

- Wire delay 5mm trace (RC model)
- SIA prediction, 1999

Technology "Improvements"

Span (bits reachable in one cycle)
[M bits, log scale]
**Technology “Improvements”**

Span (Fraction of the chip area reachable in one cycle)

- Clock = 16 gate delays
- Clock = 8 gate delays

Relative CPU Performance (log scale)

- Historical rate: 55%/y
- Business as usual (Same arch.): -12%/y
- 0.6 - 7.4x

**So, what will the performance be?**

- 2 cycles: 16kB
- 40 cycles: 1 MB
- 100 cycles: $32MB
- 400 cycles: 1GB

**Can we continue to make large and complicated CPU?...**

- Huge monolithic single-thread monsters
- Diminishing return on...
  - pipeline stages
  - parallel pipelines
- Smaller reach ➔
  - hard to add complexity, smaller caches
- Memory is often the bottleneck anyhow

**Not according to the technology forecast**

See the ISCA paper for details...
CPU Future: there is hope

- Have you ever heard of parallel threads?
- Thread-level parallelism (TLP)
  → Multiple “threads” per CPU chip
    ... SMT (Simultaneous Multi Threading)
    ... CMP (Chip Multi Processor)

SMT: Simultaneous Multithreading
“Combine TLP&ILP to find independent instr.”

+ Single-thread performance

- Multi-thread performance
- Still one monolythic pipe

CMP: Chip Multiprocessor
more TLP & geographical locality

+ Simple!!
+ Maximizes geographical locality
- Single-thread performance

CMP: Chip Multiprocessor
“Shared Memory”

Simple fast CPU core (w/ SMT?)
+ local caches
The future is here today!

- Intel XEON: 2x SMT “hyperthreading”
- IBM Power4: 2x CMP
- Sun: Rumors has it...
- IA64: Maybe 2005

The Computer Shrunk Again

Old Mainframes

Super Minis:

Microprocessor:

Chip Multiprocessor (CMP):

... and so did the large server