CPU design options

Erik Hagersten
Uppsala University

DARK2 in a nutshell

1. Memory Systems (caches, VM, DRAM, microbenchmarks, ...)
2. Multiprocessors (TLP, coherence, interconnects, scalability, clusters, ...)
3. CPUs (pipelines, ILP, scheduling, Superscalars, VLIWs, embedded, ...)
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
- BARK, KTH, Gösta Neovius, Electro-mechanic early 50s
- BESK, KTH, Erik 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 \(\Rightarrow\) performance
- \(\text{CPI} = \text{CPI}_{\text{CPU}} + \text{CPI}_{\text{Mem}}\)

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

\[
\text{Speedup} = \frac{\text{Execution time OLD}}{\text{Execution time NEW}} = \frac{\left[(1 - \text{Fraction ENHANCED}) + \frac{\text{Fraction ENHANCED}}{\text{Speedup ENHANCED}}\right]}{\text{Speedup ENHANCED}}
\]

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

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

ISA 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

Compilers – a moving target!
The impact of compiler optimizations

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

Memory allocation model also has a huge impact

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

Execution in a CPU

- "Machine Code"
- "Data"
- CPU
### Operand models

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

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</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:

<table>
<thead>
<tr>
<th>Mem</th>
<th>Stack</th>
<th>Accumulator</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B:14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C:10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Stack-based machine

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

Mem:

<table>
<thead>
<tr>
<th>Mem</th>
<th>Stack</th>
<th>Accumulator</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B:14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C:10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Stack-based machine

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

Mem:

<table>
<thead>
<tr>
<th>Mem</th>
<th>Stack</th>
<th>Accumulator</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B:14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C:10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Stack-based machine

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

Mem:

<table>
<thead>
<tr>
<th>Mem</th>
<th>Stack</th>
<th>Accumulator</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B:14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C:10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stack-based machine

Example: C := A + B

Mem:

<table>
<thead>
<tr>
<th>Mem</th>
<th>A:12</th>
<th>B:14</th>
<th>C:10</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>14</td>
<td>26</td>
<td></td>
</tr>
</tbody>
</table>

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

Register-based machine
Example: \( C := A + B \)

Data:

\[
\begin{align*}
A &: 12 \\
B &: 14 \\
C &: 26
\end{align*}
\]


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!!

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
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%)

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!
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.

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)</td>
<td>Special bits are manipulated</td>
<td>CC set for free</td>
<td>Extra state</td>
</tr>
<tr>
<td>Condition register</td>
<td>Test general simple purpose register</td>
<td>Uses up registers</td>
<td></td>
</tr>
<tr>
<td>Compare and branch</td>
<td>Compare is part of branch</td>
<td>One instr. instead of two</td>
<td>Extra work per instr.</td>
</tr>
</tbody>
</table>

Instruction formats

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

Example: DLX - A generic architecture

Load/store architecture (32 bits)

- Many (32) general purpose 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

**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>
<tr>
<td>Control</td>
<td>BEQZ R1,KALLE</td>
<td>if (Regs[R1]==0) PC ← KALLE + 4</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!)
- `0^8`: A byte of all zeroes (repeat the field `n` times)
- `M[40]`: byte 40 in memory
- `0^8 ## M[40]`: Concatenate zero-byte (MSB) with the byte @mem(40)

### Examples Hardware Descriptions

- **LBU R1, 40(R3)** -- load byte unsigned
  ```plaintext
  R1 ← 32 024 ## M[40+R3]
  ```
  ```plaintext
<table>
<thead>
<tr>
<th>8</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 31</td>
<td></td>
</tr>
<tr>
<td>SSSSSSSSSSSSSSSS</td>
<td>M[40+R3]</td>
</tr>
<tr>
<td>M[40+R3]</td>
<td>M[41+R3]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
  R1:
  ```

- **LH R1, 40(R3)** -- load halfword (signed)
  ```plaintext
  R1 ←-32 (M[40+R3]0)16 ## M[40+R3] ## M[41+R3]
  ```
  ```plaintext
<table>
<thead>
<tr>
<th>32</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 31</td>
<td></td>
</tr>
</tbody>
</table>
  | SSSSSSSSSSSSSSSSSSS | M[40+R3]
  | M[40+R3] | M[41+R3] |
  |
  R1:
  ```

- **LF, SF**: word chunks to floating point regs
- **LD, SD**: double precision to FP regs (2 regs per OP)
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

Examples Hardware Descriptions (2)

```
LHI R1, #42 -- load high immediate
R1 <-- 32 "42" #016
```

```
ADDI R1, R2, #6 -- add immediate
R1 <-- 32 R3 + "6"
```

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

<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>

<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>

<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>
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_{16}
  - BFPT, BFPF, cmp to “FP compare bit”,
    PC := PC+4+immediate_{16}
- Jumps
  - J: Jump -- 
    PC := PC + immediate_{26}
  - JAL: Jump And Link -- 
    R31 := PC+4; PC := PC + immediate_{26}
  - JALR: Jump And Link Register -- 
    R31 := PC+4; PC := PC + Reg
  - JR: Jump Register -- 
    PC := PC + Reg ("return from JAL or JALR")

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]

Ifetch

I R X W

Regs

Mem

Store Operation:

ST mem[cnst+R1], R2

Ifetch

A

I R X W

Regs

Mem

EXAMPLE: Branch to R2 if R1 == 0

BEQZ R1, R2

Op:
R1 == 0?

Initially

D
IF RegC < 100 GOTO A
C
RegC := RegC + 1
B
RegB := RegA + 1
A
LD RegA, (100 + RegC)
Cycle 5

LD 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)

Cycle 8

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

IF | ID | EX | M | WB

Example: 5-stage pipeline

IF | ID | EX | M | WB

Example: 5-stage pipeline

IF | ID | EX | M | WB

Example: 5-stage pipeline

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

Pending sequence  \( O_{p_i} A \)

RAW (Read-After-Write)
\( O_{p_i+1} \) reads A before \( O_{p_i} \) modifies A. \( O_{p_i+1} \) reads old A!

WAR (Write-After-Read) \( O_{p_i+1} \) modifies A before \( O_{p_i} \) reads A. \( O_{p_i} \) reads new A

WAW (Write-After-Write) \( O_{p_i+1} \) modifies A before \( O_{p_i} \). The value in A is the one written by \( O_{p_i} \), 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)
**Fix alt1: code scheduling**

Swap!!

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

**Fix alt2: Bypass hardware**

- **IF**
- **ID**
- **EX**
- **M**
- **WB**

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

**DLX with bypass**

**Branch delays**

- **PC**
- **A**: "Stall"
- **B**: "Stall"
- **C**: "Stall"
- **D**: IF RegC < 100 GOTO A
- **RegB := RegA + 1**
- **RegC := RegC + 1**
- **LD RegA, (100 + RegC)**

8 cycles per iteration of 4 instructions 😊

Need longer basic blocks with independent instr.
Avoiding control hazards

- Branch condition and target addr. needed here
- Branch condition and target addr. available here

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

- Move Earlier

Fix2: Static tricks

- Predict Branch not taken (a fairly rare case)
  - Define branch to take place after a following instruction
  - "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.

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)
ADD F4, F0, F2
SD 0(R1), F4
SUBI R1, R1, #8
BNEZ R1, loop

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

Can we do better by scheduling across iterations?

Statically scheduled loop

Loop:

LD F0, 0(R1)
ADD F4, F0, F2
SUBI R1, R1, #8
BNEZ R1, loop

5 instructions + 1 bubble = 6 cycles / iteration

Can we do even better by scheduling across iterations?

Optimized scheduled unrolled loop

Loop:

LD F0, 0(R1)
ADD F4, F0, F2
SUBI R1, R1, #8
BNEZ R1, loop

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.

**Example:**

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

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

- Iteration i
- Iteration i+1
- Iteration i+2

Software pipelined Loop 1

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Iteration 1</th>
<th>Iteration 2</th>
<th>Iteration 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNEQ</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Software pipelining 2(3)**

Example:

```plaintext
loop: SD 0(R1),F4  ADDD F4,F0,F2  LD F0,0(R1)
```

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

- Iteration i
- Iteration i+1
- Iteration i+2

Software pipelined Loop 2

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Iteration 1</th>
<th>Iteration 2</th>
<th>Iteration 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNEQ</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Software pipelining 3(3)**

Instructions from three consecutive iterations form the loop body:

```plaintext
loop: SD 0(R1),F4  ADDD F4,F0,F2  SD 0(R1),F4  ADDD F4,F0,F2  SD 0(R1),F4  BNEZ R1,loop
```

**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
    If \( (X > 0) \) 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
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>
</tbody>
</table>

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

Statically Scheduled Superscalar DLX

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

- Can be scheduled dynamically with Tomasulo's alg.
- Issue: Difficult to find a sufficient number of instr. to issue
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>NOP</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

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

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 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)

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

DIV $F0, F2, F4
ADDD $F6, $F0, $F8
WAR $F6, $F0, $F8
WAW $F6, $F10, $F14
SUBD $F8, $F10, $F14
RAW $F8, $F10, $F8
MULD $F6, $F10, $F8

WAR and WAW avoided through “register renaming”

Register Renaming:
DIV $F0, $F2, $F4
ADDD $F6, $F0, $F8
SUBD $F8, $F10, $F14
MULD $F6, $F10, $F8

Tomasulo’s Algorithm

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

Simple Tomasulo’s Algorithm
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
Tomasulo’s: What is going on?

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

Dynamic Scheduling Past Branches

- 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

Pipeline stage | Problem causing exception
--- | ---
IF | Page fault on instruction fetch; misaligned memory access; memory protection violation
ID | Undefined or illegal opcode
EX | Arithmetic exception
MEM | Page fault on data access; misaligned memory access; memory protection violation
WB | none

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

FP Exceptions

Example:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVF F0,F2,F4</td>
<td>24 cycles</td>
</tr>
<tr>
<td>ADDF F10,F10,F8</td>
<td>3 cycles</td>
</tr>
<tr>
<td>SUBF F12,F12,F14</td>
<td>3 cycles</td>
</tr>
</tbody>
</table>

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

A pipeline implements precise interrupts iff:

All instructions before the faulting instruction can complete

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

ROB helps the implementation in O-O-O execution

VLIW: Very Long Instruction Word

VLIW: 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 -6(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;
```

```
Non speculative code Speculative code
LW R1,0(R3) LW R1,0(R3)
BNEZ R1,L1 LW R14,0(R2)
LW R1,0(R2) BEQZ R1,L3
J L2 ADD R14,R14
L1: ADD R1,R1,4 L3: SW 0(R3),R14
L2: SW 0(R3),R1
```

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

- What about exceptions?
Example: Moving LD above a branch

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

Good performance if the branch is not taken

Example: Moving LD above a ST

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

Using Predicates

CGT R3,R1,R2
BRNZ R3, elseLD 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

Standard Technique

One instruction sets the two predicate Regs
P6: LD R7, 100(R1)
P6: ADD R1, R1, #1
P7: LD R7, 100(R2)
P7: ADD R2, R2, #1

IF R1 > R2 then P6=1;P7=0
else P6=0;P7=1

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

<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)
  - 63
- Physical Regs
  - 127
- Dynamic register window for GPRs
  - Explicit Regs (seen by main)
    - 63
    - Global: 31
    - Unused: 63
  - Physical Regs
    - 127
    - Global: 31
    - Dynamic main: 63
Calling Procedure A

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

Explicit Regs
Physical Regs

Explicit Regs

(seen by main) (seen by proc A)

Output
Input

10
31
31

31

127
85

Procedure!!!

Calling Procedure B

(automatic passing of parameters)

Explicit Regs
Physical Regs

Explicit Regs

(seen by main) (Proc A) (Proc B)

Output
Input

shared

63

10

54

54

63

4

10

85

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 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”)