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 Delay Storage Automatic Calculator
- Mark-I... H. Aiken, Harvard, WW2, Electro-mechanic
- K. Zuse, Germany, electromech. computer, special purpose, WW2
- BARK, KTH, Gösta Neovius (was at Ericsson), Electro-mechanic, early 50s
- BESK, KTH, Erik Stemme (was at Chalmers) early 50s
- SMIL, LTH mid 50s

How do you tell a good idea from a bad

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

**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"
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>ADD ([B])</td>
</tr>
<tr>
<td>PUSH ([B])</td>
<td>ADD R1,([A])</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:

\( \text{A:12, B:14, C:10} \)

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

Example: $C := A + B$

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

Stack-based

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

Accumulator-based

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

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

Register-based machine

Example: $C := A + B$

```
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 (1instr. = 4bytes)
- 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
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>Load/store (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>

Instruction formats

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

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>

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 Instruction Formats

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

- Branches if equal or if not equal
  - BEQZ, BNEZ, cmp to register,
    \( PC := PC + 4 + \text{immediate}_{16} \)
  - BFPT, BFNF, cmp to "FP compare bit",
    \( PC := PC + 4 + \text{immediate}_{16} \)
- Jumps
  - J: Jump --
    \( PC := PC + 4 + \text{immediate}_{16} \)
  - JAL: Jump And Link --
    \( R31 := PC + 4; PC := PC + \text{immediate}_{26} \)
  - JALR: Jump And Link Register --
    \( R31 := PC + 4; PC := PC + \text{Reg} \)
  - JR: Jump Register --
    \( PC := PC + \text{Reg} \) ("return from JAL or JALR")

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>
<tbody>
<tr>
<td>Immediate</td>
<td>ADD R3, R4, #3</td>
<td>Reg(R3) ← Reg(R4) + 3</td>
<td>For constants.</td>
</tr>
<tr>
<td>Displacement</td>
<td>Add R3, R4, 100(R1)</td>
<td>Mem(100 + Reg(R1))</td>
<td>Accessing local variables.</td>
</tr>
</tbody>
</table>

Size of immediates

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

EXAMPLE: pipeline implementation

Add R1, R2, R3

Implementing ISAs --pipelines

Erik Hagersten
Uppsala University
Load Operation:

**LD R1, mem[cnst+R2]**

<table>
<thead>
<tr>
<th>A</th>
<th>+</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifetch</td>
<td>R&lt;sub&gt;1&lt;/sub&gt;, R&lt;sub&gt;2&lt;/sub&gt;</td>
<td>Mem</td>
</tr>
</tbody>
</table>

Store Operation:

**ST mem[cnst+R1], R2**

<table>
<thead>
<tr>
<th>A</th>
<th>+</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifetch</td>
<td>R&lt;sub&gt;1&lt;/sub&gt;, R&lt;sub&gt;2&lt;/sub&gt;</td>
<td>Mem</td>
</tr>
</tbody>
</table>

EXAMPLE: Branch to R2 if R1 == 0

**BEQZ R1, R2**

<table>
<thead>
<tr>
<th>A</th>
<th>+</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifetch</td>
<td>Reg&lt;sub&gt;1&lt;/sub&gt; == 0?</td>
<td>Mem</td>
</tr>
</tbody>
</table>

Initially

- IF Reg<sub>C</sub> < 100 GOTO A
- Reg<sub>B</sub> := Reg<sub>C</sub> + 1
- Reg<sub>B</sub> := Reg<sub>A</sub> + 1
- LD Reg<sub>A</sub>, (100 + Reg<sub>C</sub>)

PC

Cycle 1

- IF Reg<sub>C</sub> < 100 GOTO A
- Reg<sub>C</sub> := Reg<sub>C</sub> + 1
- Reg<sub>B</sub> := Reg<sub>A</sub> + 1
- LD Reg<sub>A</sub>, (100 + Reg<sub>C</sub>)

PC

Cycle 2

- IF Reg<sub>C</sub> < 100 GOTO A
- Reg<sub>C</sub> := Reg<sub>C</sub> + 1
- Reg<sub>B</sub> := Reg<sub>A</sub> + 1
- LD Reg<sub>A</sub>, (100 + Reg<sub>C</sub>)

PC
Cycle 3

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

Cycle 4

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

Cycle 5

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

Cycle 6

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

Cycle 7

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

Cycle 8

PC ➔
IF RegC < 100 GOTO A
LD RegA, (100 + RegC)
Example: 5-stage pipeline

Example: 5-stage pipeline

Example: 5-stage pipeline

Example: 5-stage pipeline

Fundamental limitations

Hazards prevent instructions from executing in parallel:

- **Structural hazards**: Simultaneous use of same resource
  - If unified I+D$: LW will conflict with later I-fetch
- **Data hazards**: Data dependencies between instructions
  - LW R1, 100(R2) /* result avail in 2 - 100 cycles */
  - ADD R5, R1, R7
- **Control hazards**: Change in program flow
  - BNEQ R1, #OFFSET
  - ADD R5, R2, R3
- **Serialization of the execution by stalling the pipeline**
  - is one, although inefficient, way to avoid hazards

Fundamental types of data hazards

- **RAW (Read-After-Write)**
  - Op+1 reads A before Op modifies A. Op+1 reads old A!
- **WAR (Write-After-Read)**
  - Op+1 modifies A before Op reads A. Op reads new A
- **WAW (Write-After-Write)**
  - Op+1 modifies A before Op. The value in A is the one written by Op, 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

Fix alt1: code scheduling

Fix alt2: Bypass hardware

DLX with bypass
Branch delays

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

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 prediction and code scheduling can reduce the branch penalty

Fix1: Minimizing Branch Delay Effects

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

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.
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
      ADD 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

```
loop: LD F0, 0(R1)
      ADD F4, F0, F2
      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 even better by scheduling across iterations?

Unoptimized loop unrolling 4x

```
loop: LD F0, 0(R1)
      ADD F4, F0, F2
      ADD 0(R1), F4
      LD F6, #8(R1)
      ADD F8, F6, F2
      SUBI R1, R1, #8
      BNEZ R1, loop
```

5 instructions + 4 bubbles = 9 cycles / iteration

Can we do even better by scheduling across iterations?

```
loop: LD F0, 0(R1)
      ADD F4, F0, F2
      ADD 0(R1), F4
      LD F6, #8(R1)
      ADD F8, F6, F2
      SUBI R1, R1, #8
      BNEZ R1, loop
```

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

All penalties are eliminated. CPI=1

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

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

**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:

**Software pipelining 3(3)**

Instructions from three consecutive iterations form the loop body:

- 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)

**Dependencies: Revisited**

Two instructions must be independent in order to execute in parallel

- Three classes of dependencies that limit parallelism:
  - Data dependencies
    - \( X := \ldots \Rightarrow X \ldots \)
  - Name dependencies
    - \( \ldots \Rightarrow X \Rightarrow \ldots \)
  - 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

Statically Scheduled Superscalar DLX

Can be scheduled dynamically with Tomasulo’s alg.

Limits to superscalar execution

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

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

Simple superscalars relying on compiler instead of HW complexity ➔ VLIW
Very Long Instruction Word (VLIW)

Compiler is responsible for instruction scheduling

<table>
<thead>
<tr>
<th>Instruction</th>
<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 F16, R1</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>LD F16, R2</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ADD F16, F14</td>
<td>ADD F16, F9, F2</td>
<td>NOP</td>
<td>NOP</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD F16, F14</td>
<td>ADD F16, F9, F2</td>
<td>NOP</td>
<td>NOP</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>ADD F20, F18</td>
<td>ADD F20, F13, F2</td>
<td>NOP</td>
<td>NOP</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD F20, F18</td>
<td>ADD F20, F13, F2</td>
<td>NOP</td>
<td>NOP</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>SD R1, F4</td>
<td>SD R1, F6</td>
<td>NOP</td>
<td>NOP</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD R1, F8</td>
<td>SD R1, F8</td>
<td>NOP</td>
<td>NOP</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SD R1, F12</td>
<td>SD R1, F12</td>
<td>NOP</td>
<td>NOP</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD F12, F4</td>
<td>ADD F12, F8</td>
<td>NOP</td>
<td>NOP</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD F12, F8</td>
<td>ADD F12, F8</td>
<td>NOP</td>
<td>NOP</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB R1, R1</td>
<td>NOOP</td>
<td>NOP</td>
<td>NOP</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNEZ R1, LOOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Predict next PC

PC: IF RegC < 100 GOTO A
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

Last 3 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

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

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...)

Overlapping Execution

Parallelism between integer and FP instructions

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:
- stalls until DIF reaches the MEM stage, or
- disabling the write to register F0 for the DIVF instruction

WAR Example:

DIVF    F8/F2,F4
FP divide 24 cycles

SUBF    F0,F8,F10
FP sub 3 cycles

SUBF 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 F0,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

Extended Scoreboard

**Issue**: Instruction is issued when:
- No structural hazard for a functional unit
- No WAW with an instruction in execution

**Read**: Instruction reads operands when they become available (RAW)

**EX**: Normal execution

**Write**: Instruction writes when all previous instructions have read or written this operand (WAW, WAR)

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

Limitations with scoreboards

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

Tomasulo's algorithm addresses the last two limitations

A more complicated example

DIV F0,F2,F4 ; delayed a long time
ADDD F6,F0,F8
WAR F9,F10,F14
MULD F6,F10,F7

WAR and WAW avoided through "register renaming"

Register Renaming:
- DIV F0,F2,F4
- ADDO F6,F0,F8
- SUBD tmp1,F10,F14 ; can be executed right away
- MULD tmp2,F10,tmp1 ; 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…)
Simple Tomasulo’s Algorithm

Tomasulo’s: What is going on?
1. Read Register:
   - Rename DestReg to the ROB location
2. Wait for all dependencies at ROB
3. After Execution
   a) Put result in ROB
   b) Broadcast result on Data Bus (CDB) to all waiting instructions
   c) Rename DestReg to the ROB location
4. When all preceding instructions have arrived at ROB:
   - Write value to DestReg

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
#3 DIV F0,F2,F4
#4 ADDD F6,F0,F8
#5 SUBD F8,F10,F14
#6 MULD F6,F10,F8
```
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
4. When all preceding 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

Wrong Prediction!!

Do not commit!
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
Example sequence:
\[
\text{lw} \quad (\text{e.g., page fault in MEM}) \\
\text{add} \quad (\text{e.g., page fault in IF})
\]

FP Exceptions
Example:  
\[
\begin{align*}
\text{DIVF } & F0,F2,F4 & 24 \text{ cycles} \\
\text{ADDF } & F10,F10,F8 & 3 \text{ cycles} \\
\text{SUBF } & F12,F12,F14 & 3 \text{ cycles}
\end{align*}
\]
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

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>SH op branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD(R0,R1)</td>
<td>LD(R0,R1)</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>1</td>
</tr>
<tr>
<td>LD(R10,R0)</td>
<td>LD(R7,R4)</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>2</td>
</tr>
<tr>
<td>LD(R10,R0)</td>
<td>LD(R2,R4)</td>
<td>ADDD F4,F0,F2</td>
<td>ADDD F0,F4,F2</td>
<td>NOP</td>
<td>3</td>
</tr>
<tr>
<td>LD(R26,-48(R1)</td>
<td>NOP</td>
<td>ADDD F10,-16(R1)</td>
<td>ADDD F12,-16(R1)</td>
<td>NOP</td>
<td>4</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>ADDD F14,-24(R1)</td>
<td>ADDD F16,-16(R1)</td>
<td>NOP</td>
<td>5</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>ADDD F22,-40(R1)</td>
<td>ADDD F24,-16(R1)</td>
<td>NOP</td>
<td>6</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>ADDD F26,-48(R1)</td>
<td>ADDD F28,-32(R1)</td>
<td>NOP</td>
<td>7</td>
</tr>
<tr>
<td>SD R(R1),F4</td>
<td>SD R(R1),F8</td>
<td>ADDD F28,F26,F2</td>
<td>ADDD F26,F28,F2</td>
<td>NOP</td>
<td>8</td>
</tr>
<tr>
<td>SD R(R1),F4</td>
<td>SD R(R1),F8</td>
<td>SD R(R1),F8</td>
<td>SD R(R1),F8</td>
<td>NOP</td>
<td>9</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>BNEZ R1,LOOP</td>
<td>10</td>
</tr>
</tbody>
</table>

VLIW: Very Long Instruction Word

- Difficult to exploit parallelism
  - N functional units and K "dependent" pipeline stages implies N*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:

Non speculative code

```
LW R1(R0)
BNEZ R1(R0)  
```

Speculative code

```
LW R1(R0)
Move past BR    
```

What about exceptions?
Speculative instructions

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

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

Example: Moving LD above a branch

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

Good performance if the branch is not taken

Example: Moving LD above a ST

```
LD.a R1, 100(R2); "advanced LD"
....
ST R7, 50(R3) ; invalidate entry if ALAT addr match
....
LD.c R1 ; Redo LD if entry in ALAT invalid
....
ALAT (advanced load address table) is an associative data structure storing tuples of: <addr, dest-reg>
```

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

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

Conditional execution

- Removes the need for some branches☺
- Conditional Instructions
  - Conditional register move
  - Compare-and-swap (atomic memory operations later)
  - 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

```
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:
```

Standard Technique

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

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

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

Itanium Registers

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

Dynamic register window

Physical Regs

Explicit Regs (seen by the instructions)
Dynamic register window for GPRs

Calling Procedure A

Calling Procedure B (automatic passing of parameters)

Register Stack Engine (RSE)

Register rotation: FP and GPRs

What is the alternative?