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,...)

Dept of Information Technology | www.it.uu.se
© Erik Hagersten | www.docs.uu.se/~eh

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

Dept of Information Technology | www.it.uu.se
© Erik Hagersten | www.docs.uu.se/~eh

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...

Dept of Information Technology | www.it.uu.se
© Erik Hagersten | www.docs.uu.se/~eh

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

Dept of Information Technology | www.it.uu.se
© Erik Hagersten | www.docs.uu.se/~eh

Two guiding stars
-- the RISC approach:

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

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

Dept of Information Technology | www.it.uu.se
© Erik Hagersten | www.docs.uu.se/~eh
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 at all that much (market-driven)
• How many of you will ever design an ISA?
• How many ISAs will be designed in Sweden?

Compiler Organization

Compilers – a moving target!
The impact of compiler optimizations
• Compiler optimizations affect the number of instructions as well as the distribution of executed instructions (the instruction mix)

Memory allocation model also has a huge impact
• **Stack**
  • local variables in activation record
  • addressing relative to stack pointer
  • stack pointer modified on call/return
• **Global data area**
  • large constants
  • global static structures
• **Heap**
  • dynamic objects
  • often accessed through pointers

Execution in a CPU
### Operand models

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

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH [A]</td>
<td>LOAD [A]</td>
<td>ADD R1,[A]</td>
</tr>
<tr>
<td>PUSH [B]</td>
<td>ADD [B]</td>
<td>STORE [C]</td>
</tr>
<tr>
<td>ADD</td>
<td>STORE [C],R1</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 \[C\]
### Stack-based machine

Example: $C := A + B$

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

### Stack-based

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

### Accumulator-based

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

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

### Register-based machine

Example: $C := A + B$

- Commercial success:
  - CISC: X86
  - RISC: (Alpha), SPARC, (HP-PA), Power, MIPS, ARM
  - VLIW: IA64
- Explicit operands (i.e., “registers”)
- Wasteful instr. format (1 instr. = 4 bytes)
- Suits optimizing compilers
- Optimal for speed!!!

### Register-based

```
Data:
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
### Instruction formats

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

### 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, 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</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 purpose register</td>
<td>Users 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>

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>

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)

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

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

**J-type**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs</th>
<th>Rd</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>31</td>
</tr>
</tbody>
</table>

**R-type**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Rs1</th>
<th>Rs2</th>
<th>Rd</th>
<th>Func</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>31</td>
</tr>
</tbody>
</table>

**J-type**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Offset added to PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</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
  - CVTF21, CVTF2D, CVTI2D, ...

### Simple Control

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

### Implementing ISAs

---pipelines

Erik Hagersten
Uppsala University

### EXAMPLE: pipeline implementation

**Add R1, R2, R3**

- Ifetch
- \( OP: + \)

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

### Load Operation:

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

- Ifetch
**Store Operation:**

\[ \text{ST mem[const+R1], R2} \]

**EXAMPLE: Branch to R2 if R1 == 0**

\[ \text{BEQZ R1, R2} \]

---

**Initially**

- **Cycle 1**
  - \[ \text{LD RegA, (100 + RegC)} \]
  - **RegC < 100 GOTO A**
  - \[ \text{RegB := RegA + 1} \]
  - \[ \text{RegC := RegC + 1} \]

**Cycle 2**

- **Cycle 3**
  - \[ \text{LD RegA, (100 + RegC)} \]
  - **RegC < 100 GOTO A**
  - \[ \text{RegB := RegA + 1} \]
  - \[ \text{RegC := RegC + 1} \]

---
**Cycle 4**

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

IF Mem

**Cycle 5**

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

IF Mem

**Cycle 6**

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

IF Mem

**Cycle 7**

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

IF Mem → Branch → Next PC

**Cycle 8**

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

IF Mem

**Example: 5-stage pipeline**

Diagram showing a 5-stage pipeline with stages: IF, ID, EX, M, WB.
Example: 5-stage pipeline

Fundamental limitations

Hazard avoidance techniques

Fundamental types of data hazards

Static techniques (compiler): code scheduling to avoid hazards

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

Hybrid techniques: rely on compiler as well as hardware techniques to resolve hazards (e.g., VLIW support – later)
Cycle 3

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

+ → IRXW Regs

Mem

Fix alt1: code scheduling

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

IRXW Regs

Mem

Fix alt2: Bypass hardware

IF → ID → EX → M → WB

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

DLX with bypass

Data$ Data$

DTLB DTLB

L2$ L2$

Mem

Instr$ Instr$

ITLB ITLB

L2$ L2$

Mem

Branch delays

PC → IRXW "Stall" Next PC
"Stall" IF RegC < 100 GOTO A
RegB := RegA + 1
"Stall" RegC := RegC + 1
LD RegA, (100 + RegC)

Branch → Next PC

Mem

Avoiding control hazards

IF → ID → EX → M → WB

Branch condition and target address needed here
Branch prediction and code scheduling can reduce the branch penalty

Duplicate resources in ALU to compute branch condition and branch target address earlier
Branch delay cannot be completely eliminated
Taking a Branch

PC := PC + imm

Fix1: Minimizing Branch Delay Effects

Fix2: Static tricks

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

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

Static scheduling to avoid stalls

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

```plaintext
for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + 10;
```

Iterations are independent => parallel execution

#### Scheduling in each loop iteration

**Original loop**

```plaintext
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
```

5 instructions + 4 bubbles = 9 cycles / iteration

**Can we eliminate all penalties in each iteration?**

**How about moving SD down?**

**Statically scheduled loop**

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

5 instructions + 1 bubble = 6 cycles / iteration

**Can we do even better by scheduling across iterations?**

**Unoptimized loop unrolling 4x**

```plaintext
loop:  LD  F0, 0(R1)
       ADDD F4, F0, F2
       SD 0(R1), F4
       ADDD 0(R1), F4
       SD 0(R1), F4
       SUBI R1, R1, #8
       ADDD F10, F2
       BNEZ R1, loop
       SD 0(R1), F10
```

24 cycles / 4 iterations = 6 cycles / iteration

**Optimized scheduled unrolled loop**

**Important steps:**

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

**Benefits of loop unrolling:**

- Provides a larger seq. instr. window (larger basic block)
- Simplifies for static and dynamic methods to extract ILP

**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 Pipelined Loop 1](image-url)
Software pipelining 2.3

Example:

```markdown
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:

```markdown
LD F0,0(R1) ; Iteration i
ADDD F4,F0,F2
SD 0(R1),F4
LD F0,0(R1) ; Iteration i+1
ADDD F4,F0,F2
SD 0(R1),F4
ADDD F4,F0,F2
SD 0(R1),F4
```

Software pipelining 3.3

Instructions from three consecutive iterations form the loop body:

```markdown
loop: SD 0(R1),F4         ; from iteration i
ADDD F4,F0,F2         ; from iteration i+1
LD F0,-16(R1)     ; from   iteration i+2
SUBI R1,R1,#8
BNEZ R1,loop
```

- 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
    - $X := \ldots$ $\ldots := X \ldots$
  - Name dependencies
    - $\ldots := X$ $X := \ldots$
  - Control dependencies
    - If $(X > 0)$ then
      - $Y := \ldots$

Getting desperate for ILP

Erik Hagersten
Uppsala University
Sweden
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

Type Pipe_stages

Int. IF ID EX MEM WB
FP IF ID EX MEM WB
Int. IF ID EX MEM WB
FP IF ID EX MEM WB

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

Statically Scheduled Superscalar DLX

- Difficult to find a sufficient number of instr. to issue
- 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

VLIW

Very Long Instruction Word (VLIW)

Compiler is responsible for instruction scheduling

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

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

Branch ➔ Next PC

Mem

Cycle 4

Guess the next PC here!!

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

Branch history table

A simple branch prediction scheme

PC:

index

1=taken

0=not taken

- 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

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

Dynamic Scheduling Of Branches

LD ADD SUB ST

LD ADD SUB ST

LD ADD SUB ST

Last 3 branches:
**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

**Return address stack**

- Popular subroutines are called from many places in the code.
- Branch prediction may be confused!!
- May hurt other predictions
- New approach:
  - Push the return address on a [small] stack at the time of the call
  - Pop addresses on return
Overlapping Execution

Erik Hagersten
Uppsala University
Sweden

Multicycle operations in the pipeline (floating point)
(Not a SuperScalar...)

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

Parallelism between integer and FP instructions

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

Dynamic Instruction Scheduling

Key idea: allow subsequent independent instructions to proceed
DIVD F0,F2,F4 ; takes long time
ADDD F0,F10,F8 ; stalls waiting for F0
SUBD F12,F8,F13 ; Let this instr. bypass the ADDD
- Enables out-of-order execution (& out-of-order completion)

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

Department of Information Technology | www.it.uu.se
© Erik Hagersten | www.docs.uu.se/~eh
**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**

Limitations with scoreboards are 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
ADD F6, F0, F8
WAW F7, F10, F14
MULD F6, F10, F8
```

WAR and WAW avoided through “register renaming”

---

**Tomasulo’s Algorithm**

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

---

**Simple Tomasulo’s Algorithm**

---

**Tomasulo’s: What is going on?**

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

IF – Issue

Reg. Write

Path

Issue

Op

S2

S1

#2

D

#1

Reg. Write Path

Common Data Bus (CDB)

Write Stage

ReOrder Buffer (ROB)

Simple Tomasulo’s Algorithm

IF – Issue

Reg. Write

Path

Issue

Op

S2

S1

#2

D

#1

Reg. Write Path

Common Data Bus (CDB)

Write Stage

ReOrder Buffer (ROB)

Simple Tomasulo’s Algorithm

IF – Issue

Reg. Write

Path

Issue

Op

S2

S1

#2

D

#1

Reg. Write Path

Common Data Bus (CDB)

Write Stage

ReOrder Buffer (ROB)

Simple Tomasulo’s Algorithm

IF – Issue

Reg. Write

Path

Issue

Op

S2

S1

#2

D

#1

Reg. Write Path

Common Data Bus (CDB)

Write Stage

ReOrder Buffer (ROB)

Simple Tomasulo’s Algorithm

IF – Issue

Reg. Write

Path

Issue

Op

S2

S1

#2

D

#1

Reg. Write Path

Common Data Bus (CDB)

Write Stage

ReOrder Buffer (ROB)
Simple Tomasulo’s Algorithm

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

Dynamic Scheduling Past Branches

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

Dealing with Exceptions
Erik Hagersten
Uppsala University
Sweden
Exception handling in pipelines

Example: Page fault from TLB

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

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

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

Guaranteeing the execution order

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

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

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

FP Exceptions

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

SUBF may generate a trap before DIVF has completed!!

Revisiting Exceptions:

A pipeline implements precise interrupts iff:

1. All instructions before the faulting instruction can complete
2. 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

<table>
<thead>
<tr>
<th>Memory</th>
<th>1GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>2MB</td>
<td></td>
</tr>
</tbody>
</table>

Erik Hagersten
Uppsala University
Sweden

HW support for [static] speculation and improved ILP
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 F10 R1</td>
<td>LD F14 R1</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>LD F10-23 R1</td>
<td>LD F22-46 R1</td>
<td>ADD F4 F2 F2</td>
<td>ADD F8 F2 F2</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>NOP</td>
<td>ADD F12 F14 F1</td>
<td>ADD F12 F14 F2</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>NOP</td>
<td>ADD F18 F16 F1</td>
<td>ADD F12 F14 F2</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>NOP</td>
<td>ADD F26 F24 F1</td>
<td>ADD F12 F14 F1</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>NOP</td>
<td>ADD F26 F24 F1</td>
<td>ADD F12 F14 F1</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>NOP</td>
<td>ADD F26 F24 F1</td>
<td>ADD F12 F14 F1</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>SOP R1 R1</td>
<td>SOP R1 R1</td>
<td>SOP R1 R1</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
</tbody>
</table>

Limits to VLIW

- 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 be exploited

Example:

```plaintext
if (A == 0) then A = B; else A = A+4;
```

Non speculative code

```plaintext
LW R1,0(R3)  
BNEZ R1,L1  
LW R1,0(R2)  
JL 2 ADD R14,R14,4
```

Speculative code

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

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

```plaintext
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 

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
    - MOVQ R1, R2, R3 (move R1 to R2 if (R3 = 0))
  - Compare-and-swap (atomics memory operations later)
    - CMP R1, R2, R3; swap R1 and move(R1) if (move(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
end

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)

**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
- 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 for GPRs

Calling Procedure A
- Procedure!!!
  (not processes)
Calling Procedure B (automatic passing of parameters)

Physical Regs

Unused

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

Register Stack Engine (RSE)

- Saves and restores registers to memory on register spills
- Implemented in hardware
- Works in the background
- Gives the illusion of an unlimited register stack
- This is similar to SPARC and UCB's RISC

Register rotation: FP and GPRs

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

What is the alternative?

- VLIW was meant to simplify HW
- Itanium 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")