Superscalar Processing

740
October 31, 2012

Evolution of Intel Processor Pipelines
- 486, Pentium, Pentium Pro

Superscalar Processor Design
- Speculative Execution
- Register Renaming
- Branch Prediction

Architectural Performance

Metric
- SpecX92/Mhz: Normalizes with respect to clock speed

Sampling

<table>
<thead>
<tr>
<th>Processor</th>
<th>MHz</th>
<th>SpecInt92</th>
<th>IntAP</th>
<th>SpecFP92</th>
<th>FltAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>i386/387</td>
<td>33</td>
<td>6</td>
<td>0.2</td>
<td>3</td>
<td>0.1</td>
</tr>
<tr>
<td>i486DX</td>
<td>50</td>
<td>28</td>
<td>0.6</td>
<td>13</td>
<td>0.3</td>
</tr>
<tr>
<td>Pentium</td>
<td>150</td>
<td>181</td>
<td>1.2</td>
<td>125</td>
<td>0.8</td>
</tr>
<tr>
<td>PentiumPro</td>
<td>200</td>
<td>320</td>
<td>1.6</td>
<td>283</td>
<td>1.4</td>
</tr>
<tr>
<td>MIPS R3000A</td>
<td>25</td>
<td>16.1</td>
<td>0.6</td>
<td>21.7</td>
<td>0.9</td>
</tr>
<tr>
<td>MIPS R10000</td>
<td>200</td>
<td>300</td>
<td>1.5</td>
<td>600</td>
<td>3.0</td>
</tr>
<tr>
<td>Alpha 21164a</td>
<td>417</td>
<td>500</td>
<td>1.2</td>
<td>750</td>
<td>1.8</td>
</tr>
</tbody>
</table>

i486 Pipeline

Fetch
- Load 16-bytes of instruction into prefetch buffer

Decode1
- Determine instruction length, instruction type

Decode2
- Compute memory address
- Generate immediate operands

Execute
- Register Read
- ALU operation
- Memory read/write

Write-Back
- Update register file

Pipeline Stage Details

Fetch
- Moves 16 bytes of instruction stream into code queue
- Not required every time
  - About 5 instructions fetched at once
  - Only useful if don’t branch
- Avoids need for separate instruction cache

D1
- Determine total instruction length
  - Signals code queue aligner where next instruction begins
- May require two cycles
  - When multiple operands must be decoded
  - About 6% of “typical” DOS program
Stage Details (Cont.)

D2
- Extract memory displacements and immediate operands
- Compute memory addresses
  - Add base register, and possibly scaled index register
  - May require two cycles
  - If index register involved, or both address & immediate operand
  - Approx. 5% of executed instructions

EX
- Read register operands
- Compute ALU function
- Read or write memory (data cache)

WB
- Update register result

Data Hazards

Data Hazards
- Generated Used Handling
  ALU ALU EX-EX Forwarding
  Load ALU EX-EX Forwarding
  ALU Store EX-EX Forwarding
  ALU Eff. Address (Stall) + EX-ID2 Forwarding

Control Hazards

Control Hazards (Cont.)

Branch Not Taken
- Allow pipeline to continue.
- Total of 1 cycle for instruction
  Jump +1
  Jump +2
  Jump +3
  Target
  Fetch

Branch taken
- Flush instructions in pipe
- Begin ID1 at target.
- Total of 3 cycles for instruction
  Jump +1
  Jump +2
  Target
  Fetch
  ID1
Comparison with Our pAlpha Pipeline

Two Decoding Stages
- Harder to decode CISC instructions
- Effective address calculation in D2

Multicycle Decoding Stages
- For more difficult decodings
- Stalls incoming instructions

Combined Mem/EX Stage
- Avoids load stall without load delay slot
  - But introduces stall for address computation

Comparison to 386

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>386 Cycles</th>
<th>486 Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Store</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>ALU</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Jump taken</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>Jump not taken</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Call</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

Reasons for Improvement
- On chip cache
  - Faster loads & stores
- More pipelining

Pentium Block Diagram

(Pentium Block Diagram from Microprocessor Report 10/28/92)

Pentium Pipeline

(U-Pipe)
- Fetch & Align Instruction
  - Decode Instruction
  - Generate Control Word
  - Decode Control Word
  - Generate Memory Address
  - Access data cache or calculate ALU result
  - Write register result

(V-Pipe)
- Fetch & Align Instruction
  - Decode Instruction
  - Generate Control Word
  - Decode Control Word
  - Generate Memory Address
  - Access data cache or calculate ALU result
  - Write register result

(Microprocessor Report 10/28/92)
Superscalar Execution

Can Execute Instructions I1 & I2 in Parallel if:
- Both are "simple" instructions
  - Don’t require microcode sequencing
  - Some operations require U-pipe resources
  - 90% of SpecInt instructions
- I1 is not a jump
- Destination of I1 not source of I2
  - But can handle I1 setting CC and I2 being cond. jump
- Destination of I1 not destination of I2

If Conditions Don’t Hold
- Issue I1 to U Pipe
- I2 issued on next cycle
  - Possibly paired with following instruction

Branch Prediction

Branch Target Buffer
- Stores information about previously executed branches
  - Indexed by instruction address
  - Specifies branch destination + whether or not taken
- 256 entries

Branch Processing
- Look for instruction in BTB
- If found, start fetching at destination
- Branch condition resolved early in WB
  - If prediction correct, no branch penalty
  - If prediction incorrect, lose ~3 cycles
    - Which corresponds to > 3 instructions
- Update BTB

Superscalar Terminology

Basic
Superscalar
  - Able to issue > 1 instruction / cycle
Superpipelined
  - Deep, but not superscalar pipeline.
    - E.g., MIPS R5000 has 8 stages
Branch prediction
  - Logic to guess whether or not branch will be taken,
    and possibly branch target

Advanced
Out-of-order
  - Able to issue instructions out of program order
Speculation
  - Execute instructions beyond branch points, possibly
    nullifying later
Register renaming
  - Able to dynamically assign physical registers to
    instructions
Retire unit
  - Logic to keep track of instructions as they
    complete.

Superscalar Execution Example

Assumptions
- Single FP adder takes 2 cycles
- Single FP multiplier takes 5 cycles
- Can issue add & multiply
together
- Must issue in-order
  - (Single adder, data dependence)
    (In order)
Critical Path = 9 cycles

Data Flow

v: add $f2, $f4, $f10
w: mult $f10, $f6, $f10
x: add $f10, $f8, $f12
y: add $f4, $f6, $f4
z: add $f4, $f8, $f10

### Adding Advanced Features

**Out Of Order Issue**
- Can start y as soon as adder available
- Must hold back z until $f10$ not busy & adder available

With Register Renaming

```
v: add $f2, $f4, $f10
w: mult $f10, $f6, $f10
x: add $f10, $f8, $f12
y: add $f4, $f6, $f4
z: add $f4, $f8, $f10
```

```
v: add $f2, $f4, $f10a
w: mult $f10a, $f6, $f10a
x: add $f10a, $f8, $f12
y: add $f4, $f6, $f4
z: add $f4, $f8, $f10
```

### Pentium Pro (P6)

**History**
- Announced in Feb. '95
- Delivering in high end machines now

**Features**
- Dynamically translates instructions to more regular format
  - Very wide RISC instructions
- Executes operations in parallel
  - Up to 5 at once
- Very deep pipeline
  - 12-18 cycle latency

### Pentium Pro Block Diagram

**Dynamically Scheduled Processors**

- PC: $0x1c$<br>  Branch Cache
- $0x1c$: $b = c / 3$; issue (out-of-order)<br>  $0x18$: $z = a + 2$; issue (out-of-order)<br>  $0x14$: $y = x + 1$; can't issue<br>  $0x10$: $x = *p$; issue (cache miss)

Fetch & graduate in-order, issue out-of-order
PentiumPro Operation

Translates instructions dynamically into “Uops”
- 118 bits wide
- Holds operation, two sources, and destination

Executes Uops with “Out of Order” engine
- Uop executed when
  - Operands available
  - Functional unit available
- Execution controlled by “Reservation Stations”
  - Keeps track of data dependencies between uops
  - Allocates resources

Read-after-Write (RAW) Dependences

Also known as a “true” dependence
Example:
- S1: addq r1, r2, r3
- S2: addq r3, r4, r4

How to optimize?
- cannot be optimized away

Write-after-Read (WAR) Dependences

Also known as an “anti” dependence
Example:
- S1: addq r1, r2, r3
- S2: addq r4, r5, r1
  ... addq r1, r6, r7

How to optimize?
- rename dependent register (e.g., r1 in S2 -> r8)
  - S1: addq r1, r2, r3
  - S2: addq r4, r5, r8
  ... addq r8, r6, r7

Write-after-Write (WAW) Dependences

Also known as an “output” dependence
Example:
- S1: addq r1, r2, r3
- S2: addq r4, r5, r3
  ... addq r3, r6, r7

How to optimize?
- rename dependent register (e.g., r3 in S2 -> r8)
  - S1: addq r1, r2, r3
  - S2: addq r4, r5, r8
  ... addq r8, r6, r7
Living with Expensive Branches

Mispredicted Branch Carries a High Cost
- Must flush many in-flight instructions
- Start fetching at correct target
- Will get worse with deeper and wider pipelines

Impact on Programmer / Compiler
- Avoid conditionals when possible
  - Bit manipulation tricks
- Use special conditional-move instructions
  - Recent additions to many instruction sets
- Make branches predictable
  - Very low overhead when predicted correctly

Some Interesting Patterns

```
PPPPPPPPP
-1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
- Should give perfect prediction

RRRRRRRRR
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
- Will mispredict 1/2 of the time

N*N[PNNP]
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
- Should alternate between states No! and No?

N*P[PNPN]
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
- Should alternate between states No? and Yes?

N*N[PPNN]
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

N*P[PPNN]
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

Branch Prediction Example

```
static void loop1() {
    int i;
    data_t abs_sum = (data_t) 0;
    data_t prod = (data_t) 1;
    for (i = 0; i < CNT; i++) {
        data_t x = dat[i];
        data_t ax = ABS(x);
        abs_sum += ax;
        prod *= x;
    }
    answer = abs_sum*prod;
}
```

Loop Performance (FP)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>PPC 604 Cycles</th>
<th>PPC 604 Penalty</th>
<th>Pentium Cycles</th>
<th>Pentium Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPPPPPPP</td>
<td>13.6</td>
<td>0</td>
<td>21.1</td>
<td>0</td>
</tr>
<tr>
<td>RRRRRRRR</td>
<td>13.6</td>
<td>0</td>
<td>23.9</td>
<td>1.8</td>
</tr>
<tr>
<td>N*N[PNPN]</td>
<td>13.6</td>
<td>0</td>
<td>15.6</td>
<td>6.6</td>
</tr>
<tr>
<td>N*P[PNPN]</td>
<td>13.6</td>
<td>0</td>
<td>15.9</td>
<td>6.3</td>
</tr>
<tr>
<td>N*N[PPNN]</td>
<td>13.6</td>
<td>0</td>
<td>12.5</td>
<td>3.3</td>
</tr>
<tr>
<td>N*P[PPNN]</td>
<td>13.6</td>
<td>0</td>
<td>12.3</td>
<td>3.1</td>
</tr>
</tbody>
</table>

Observations
- 604 has prediction rates 0%, 50%, and 100%
  - Expected 50% from N*N[PNPN]
  - Expected 25% from N*N[PPNN]
- Pentium appears to be more variable, ranging 0 to 100%
- Special Patterns Can Be Worse than Random
  - Only 50% of all people are "above average"
**Loop 1 Surprises**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>R10000</th>
<th>Pentium II</th>
</tr>
</thead>
<tbody>
<tr>
<td>0*210000</td>
<td>3.1</td>
<td>0</td>
</tr>
<tr>
<td>0*210000</td>
<td>3.3</td>
<td>0</td>
</tr>
<tr>
<td>0*220000</td>
<td>3.6</td>
<td>0</td>
</tr>
<tr>
<td>0*210000</td>
<td>3.6</td>
<td>0</td>
</tr>
<tr>
<td>0*220000</td>
<td>3.6</td>
<td>0</td>
</tr>
<tr>
<td>0*210000</td>
<td>3.6</td>
<td>0</td>
</tr>
<tr>
<td>0*220000</td>
<td>3.6</td>
<td>0</td>
</tr>
</tbody>
</table>

**Pentium II**

- Random shows clear penalty
- But others do well
  - More clever prediction algorithm

**R10000**

- Has special "conditional move" instructions
- Compiler translates $a = \text{Cond} \ ? \ Texpr : Fexpr$ into
  
  $$a = Fexpr$$
  $$temp = Texpr$$
  $$\text{CMOV}(a, temp, \text{Cond})$$

  - Only valid if $Texpr \ & \ Fexpr$ can’t cause error

---

**P6 Branch Prediction**

**Two-Level Scheme**

- Yeh & Patt, ISCA ’93
- Keep shift register showing past $k$ outcomes for branch
- Use to index $2^k$ entry table
- Each entry provides 2-bit, saturating counter predictor
- Very effective for any deterministic branching pattern

---

**Branch Prediction Comparisons**

**21264 Branch Prediction Logic**

- Purpose: Predict whether or not branch taken
- 35Kb of prediction information
- 2% of total die size
- Claim 0.7 - 1.0% misprediction
Core i7 Pipeline: Big Picture

Core i7 Pipeline: Front End

Core i7 Pipeline: Execution Unit

Core i7 Pipeline: Memory Hierarchy