Superscalar Processing

CS 740
September 15-17, 2003

Intel Processors
- 486, Pentium, Pentium Pro

Superscalar Processor Design
- Use PowerPC 604 as case study
- Speculative Execution, Register Renaming, Branch Prediction

More Superscalar Examples
- MIPS R10000
- DEC Alpha 21264

How Can We Speed This Up?

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.

Issues?

- Dependencies
  - Data
  - Control
- Memory Latency
- Structural Hazards
- ?
Intel x86 Processors

<table>
<thead>
<tr>
<th>Processor</th>
<th>Year</th>
<th>Transistors</th>
<th>MHz</th>
<th>Spec92 (Int/FP)</th>
<th>Spec95 (Int/FP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8086</td>
<td>'78</td>
<td>29K</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>i286</td>
<td>'83</td>
<td>134K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i386</td>
<td>'86</td>
<td>275K</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i386</td>
<td>'88</td>
<td>33</td>
<td>6 / 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i486</td>
<td>'89</td>
<td>1.2M</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>50</td>
<td>28 / 13</td>
<td></td>
</tr>
<tr>
<td>Pentium</td>
<td>'93</td>
<td>3.1M</td>
<td>66</td>
<td>78 / 64</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>150</td>
<td>181 / 125</td>
<td>4.3 / 3.0</td>
</tr>
<tr>
<td>PentiumPro</td>
<td>'95</td>
<td>5.5M</td>
<td>150</td>
<td>245 / 220</td>
<td>6.1 / 4.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>200</td>
<td>320 / 283</td>
<td>8.2 / 6.0</td>
</tr>
<tr>
<td>Pentium II</td>
<td>'97</td>
<td>7.5M</td>
<td>300</td>
<td></td>
<td>11.6 / 6.8</td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td></td>
<td>42M</td>
<td>2000</td>
<td></td>
</tr>
</tbody>
</table>

Other Processors

<table>
<thead>
<tr>
<th>Processor</th>
<th>Year</th>
<th>Transistors</th>
<th>MHz</th>
<th>Spec92 Spec95</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS R3000</td>
<td>'88</td>
<td>25</td>
<td>16.1 / 21.7</td>
<td></td>
</tr>
<tr>
<td>(DecStation 5000/120)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPS R5000</td>
<td></td>
<td>3.6M</td>
<td>180</td>
<td>4.1 / 4.4</td>
</tr>
<tr>
<td>(Wean Hall SGI)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPS R10000</td>
<td>'95</td>
<td>5.9M</td>
<td>200</td>
<td>300 / 600 8.9 / 17.2</td>
</tr>
<tr>
<td>(Most Advanced MIPS)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpha 21164a</td>
<td>'96</td>
<td>9.3M</td>
<td>417</td>
<td>500 / 750 11 / 17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>500</td>
<td>12.6 / 18.3</td>
</tr>
<tr>
<td>Alpha 21264</td>
<td>'97</td>
<td>15M</td>
<td>500</td>
<td>30 / 60</td>
</tr>
<tr>
<td>Alpha 21364</td>
<td>'99?</td>
<td>100M</td>
<td>1200</td>
<td>70 / 120?</td>
</tr>
<tr>
<td>(last one made 192M Ts (8 for logic)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Architectural Performance

Metric
- SpecX92/Mhz: Normalizes with respect to clock speed
- But ... one measure of good arch. is how fast can run clock

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>

Performance: Ops/Clk * Clks/Sec

Specint2000
Multiple Data Sizes and Addressing Methods
- Recent generations optimized for 32-bit mode

Limited Number of Registers
- Stack-oriented procedure call and FP instructions
- Programs reference memory heavily (41%)

Variable Length Instructions
- First few bytes describe operation and operands
- Remaining ones give immediate data & address displacements
- Average is 2.5 bytes

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

ID1 | ID2 | EX
---|---|---
Fetch

Jump Instruction Processing
- Continue pipeline assuming branch not taken
- Resolve branch condition in EX stage
- Also speculatively fetch at target during EX stage

Branch Not Taken
- Allow pipeline to continue
- Total of 1 cycle for instruction

Branch taken
- Flush instructions in pipe
- Begin ID1 at target
- Total of 3 cycles for instruction
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

Cycles Per Instruction

<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 Pipeline

Fetch & Align Instruction
- Decode Instr.
- Generate Control Word

Decode Control Word
- Generate Memory Address

Access data cache or calculate ALU result

Write register result
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 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)

Data Flow

Critical Path = 9 cycles

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

FUs

Matching unit

Why did this fail?
- Think about scaling
- Think about for (i=0; i<100; i++) B[i] = cos(A[i])

Dataflow Machines

- Popular research area in 80's and 90's
- Idea is to execute at dataflow limit
- TTDA
  - Every operation tagged with sources and destinations
  - Matching unit (like a big reservation station)
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
- $v$: addt $f2$, $f4$, $f10$
- $w$: addt $f10$, $f6$, $f10$
- $x$: addt $f10$, $f8$, $f12$
- $y$: addt $f4$, $f6$, $f4$
- $z$: addt $f4$, $f8$, $f10$

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

Pentium Pro 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
Branch Prediction

Critical to Performance
- 11-15 cycle penalty for misprediction

Branch Target Buffer
- 512 entries
- 4 bits of history
- Adaptive algorithm
  - Can recognize repeated patterns, e.g., alternating taken-not taken

Handling BTB misses
- Detect in cycle 6
- Predict taken for negative offset, not taken for positive
  - Loops vs. conditionals

Limitations of x86 Instruction Set

Not enough registers
- too many memory references

Intel is switching to a new instruction set for Merced
- IA-64, joint with HP
- Will dynamically translate existing x86 binaries

PPC 604

Superscalar
- Up to 4 instructions per cycle

Speculative & Out-of-Order Execution
- Begin issuing and executing instructions beyond branch

Other Processors in this Category
- MIPS R10000
- Intel PentiumPro & Pentium II
- Digital Alpha 21264

604 Block Diagram
**General Principles**

Must be Able to Flush Partially-Executed Instructions
- Branch mispredictions
- Earlier instruction generates exception

Special Treatment of “Architectural State”
- Programmer-visible registers
- Memory locations
- Don’t do actual update until certain instruction should be executed

Emulate “Data Flow” Execution Model
- Instruction can execute whenever operands available

**Processing Stages**

Fetch
- Get instruction from instruction cache

Dispatch (~= Decode)
- Get available operands
- Assign to hardware execution unit

Execute
- Perform computation or memory operation
  - Store’s are only buffered

Retire / Commit (~= Writeback)
- Allow architectural state to be updated
  - Register update
  - Buffered store

**Fetching Instructions**

- Up to 4 fetched from instruction cache in single cycle

Branch Target Address Cache (BTAC)
- Target addresses of recently-executed, predicted-taken branches
  - 64 entries
  - Indexed by instruction address
- Accessed in parallel with instruction fetch
- If hit, fetch at predicted target starting next cycle

**Branch Prediction**

Branch History Table (BHT)
- 512 state machines, indexed by low-order bits of instruction address
- Encode information about prior history of branch instructions
  - Small chance of two branch instructions aliasing
  - Predict whether or not branch will be taken
  - 3 cycle penalty if mispredict

Interaction with BTAC
- BHT entries start in state No!
- When make transition from No? to Yes?, allocate entry in BTAC
- Deallocate when make transition from Yes? to No!
**Dispatch**

- Up to 4 instructions per cycle
  - Assign to execution units
  - Put entry in retirement buffer
  - Assign rename registers
  - Ignore data dependencies

**Dispatching Actions**

- **Generate Entry in Retirement Buffer**
  - 16-entry buffer tracking instructions currently “in flight”
    - Dispatched but not yet completed
  - Circular buffer in program order
  - Instruction tagged with branches they depend on
    - Easy to flush if mispredicted

- **Assign Rename Register as Target**
  - Additional registers (12 integer, 8 FP) used as targets for in-flight instructions
  - Instruction updates this register
  - Update of actual architectural register occurs only when instruction retired

**Hazard Handling with Renaming**

- **Dispatch Unit Maintains Mapping**
  - From register ID to actual register
  - Could be the actual architectural register
    - Not target of currently-executing instruction
  - Could be rename register
    - Perhaps already written by instruction that has not been retired
      - E.g., still waiting for confirmation of branch prediction
    - Perhaps instruction result not yet computed
      - Grab later when available

- **Hazards**
  - **RAW:** Mapping identifies operand source
  - **WAR:** Write will be to different rename register
  - **WAW:** Writes will be to different rename register

**Read-after-Write (RAW) Dependences**

- Also known as a “true” dependence

**Example:**

- S1: \( \text{addq r1, r2, r3} \)
- S2: \( \text{addq r3, r4, r4} \)

**How to optimize?**

- cannot be optimized away
Write-after-Read (WAR) Dependences

Also known as an “anti” dependence
Example:

S1: \( \text{addq } r1, r2, r3 \)
S2: \( \text{addq } r4, r5, r1 \)
   \( \ldots \)
   \( \text{addq } r1, r6, r7 \)

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

Write-after-Write (WAW) Dependences

Also known as an “output” dependence
Example:

S1: \( \text{addq } r1, r2, r3 \)
S2: \( \text{addq } r4, r5, r3 \)
   \( \ldots \)
   \( \text{addq } r3, r6, r7 \)

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

Moving Instructions Around

Reservation Stations
• Buffers associated with execution units
• Hold instructions prior to execution
  — Plus those operands that are available
• May be waiting for one or more operands
  — Operand mapped to rename register that is not yet available
• May be waiting for unit to be available

Completion Busses
• Results generated by execution units
• Tagged by rename register ID
• Monitored by reservation stations
  — So they can get needed operands
  — Effectively implements bypassing
• Supply results to completion unit

Execution Resources

Integer
• Two units to handle regular integer instructions
• One for “complex” operations
  — Multiply with latency 3--4 and throughput once per 1--2 cycles
  — Unpipelined divide with latency 20

Floating Point
• Add/multiply with latency 3 and throughput 1
• Unpipelined divide with latency 18--31

Load Store Unit
• Own address ALU
• Buffer of pending store instructions
  — Don’t perform actual store until ready to retire instruction
• Loads can be performed speculatively
  — Check to see if target of pending store operation
Retiring Instructions

Retire in Program Order
- When instruction is at head of buffer
- Up to 4 per cycle
- Enable change of architectural state
  - Transfer from rename register to architectural
  - Free rename register for use by another instruction
  - Allow pending store operation to take place

Flush if Should not be Executed
- Tagged by branch that was mispredicted
- Follows instruction that raised exception
- As if instructions had never been fetched

Execution Example

Assumptions
- Two-way issue with renaming
  - Rename registers %f0, %f2, etc.
- 1 cycle add.d latency, 2 cycle mult.d

<table>
<thead>
<tr>
<th>Value</th>
<th>Valid</th>
<th>%f0</th>
<th>%f2</th>
<th>%f4</th>
<th>%f6</th>
<th>%f8</th>
<th>%f10</th>
<th>%f12</th>
<th>%f14</th>
<th>%f16</th>
<th>%f18</th>
<th>%f20</th>
</tr>
</thead>
<tbody>
<tr>
<td>%f0</td>
<td>T</td>
<td>10.0</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%f2</td>
<td>T</td>
<td>10.0</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%f4</td>
<td>T</td>
<td>20.0</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%f6</td>
<td>T</td>
<td>40.0</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%f8</td>
<td>T</td>
<td>80.0</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%f10</td>
<td>T</td>
<td>160.0</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%f12</td>
<td>T</td>
<td>320.0</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%f14</td>
<td>F</td>
<td></td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%f16</td>
<td>F</td>
<td></td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%f18</td>
<td>F</td>
<td></td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%f20</td>
<td>F</td>
<td></td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Actual
- %f14
- %f16
- %f18
- %f20

ROB:

Op1 Op2 Dest Op1 Op2 Dest
-- -- -- -- -- -- -- --
ADD MULT ADD MULT

Actions
- Instructions v & w issued
  - v target set to %f14
  - w target set to %f16

604 Chip

- Originally 200 mm²
  - 0.65µm process
  - 100 MHz
- Now 148 mm²
  - 0.35µm process
  - bigger caches
  - 300 MHz
- Performance requires real estate
  - 11% for dispatch & completion units
  - 6% for register files
  - Lots of ports

Figure 3. The PowerPC 604 incorporates 3.6 million transistors on a 12.4 x 15.8 mm die using 0.65-micron, five-layer-metal CMOS.
### Execution Example: Cycle 2

**Actions**
- Instructions x & y issued
  - x & y targets set to %f18 and %f20
- Instruction v executed

**ROB:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Valid</th>
<th>Actual</th>
<th>OP1</th>
<th>OP2</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0</td>
<td>T</td>
<td>%f0</td>
<td></td>
<td></td>
<td>v</td>
</tr>
<tr>
<td>10.0</td>
<td>T</td>
<td>%f2</td>
<td></td>
<td></td>
<td>%f2</td>
</tr>
<tr>
<td>20.0</td>
<td>T</td>
<td>%f4</td>
<td></td>
<td></td>
<td>%f4</td>
</tr>
<tr>
<td>40.0</td>
<td>T</td>
<td>%f6</td>
<td></td>
<td></td>
<td>%f6</td>
</tr>
<tr>
<td>80.0</td>
<td>T</td>
<td>%f8</td>
<td></td>
<td></td>
<td>%f8</td>
</tr>
<tr>
<td>160.0</td>
<td>T</td>
<td>%f10</td>
<td></td>
<td></td>
<td>%f10</td>
</tr>
<tr>
<td>320.0</td>
<td>T</td>
<td>%f12</td>
<td></td>
<td></td>
<td>%f12</td>
</tr>
<tr>
<td>30.0</td>
<td>F</td>
<td>%f16</td>
<td></td>
<td></td>
<td>%f16</td>
</tr>
<tr>
<td>10.0</td>
<td>F</td>
<td>%f18</td>
<td></td>
<td></td>
<td>%f18</td>
</tr>
<tr>
<td>10.0</td>
<td>F</td>
<td>%f20</td>
<td></td>
<td></td>
<td>%f20</td>
</tr>
</tbody>
</table>

**Value**

- %f0: 40.0
- %f2: 20.0
- %f4: 40.0
- %f6: 40.0
- %f8: 40.0
- %f10: 40.0
- %f12: 40.0
- %f16: 40.0
- %f18: 40.0
- %f20: 40.0

**Operands**

- ADD: 30.0, 40.0, %f16
- MULT: 30.0, 40.0, %f16

**Actions**
- Instruction v retired
  - But doesn't change %f10
- Instruction w begins execution
  - Moves through 2 stage pipeline
- Instruction y executed
- Instruction z issued

**ROB:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Valid</th>
<th>Actual</th>
<th>OP1</th>
<th>OP2</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0</td>
<td>T</td>
<td>%f0</td>
<td></td>
<td></td>
<td>v</td>
</tr>
<tr>
<td>10.0</td>
<td>T</td>
<td>%f2</td>
<td></td>
<td></td>
<td>%f2</td>
</tr>
<tr>
<td>20.0</td>
<td>T</td>
<td>%f4</td>
<td></td>
<td></td>
<td>%f4</td>
</tr>
<tr>
<td>40.0</td>
<td>T</td>
<td>%f6</td>
<td></td>
<td></td>
<td>%f6</td>
</tr>
<tr>
<td>80.0</td>
<td>T</td>
<td>%f8</td>
<td></td>
<td></td>
<td>%f8</td>
</tr>
<tr>
<td>160.0</td>
<td>T</td>
<td>%f10</td>
<td></td>
<td></td>
<td>%f10</td>
</tr>
<tr>
<td>320.0</td>
<td>T</td>
<td>%f12</td>
<td></td>
<td></td>
<td>%f12</td>
</tr>
<tr>
<td>30.0</td>
<td>F</td>
<td>%f16</td>
<td></td>
<td></td>
<td>%f16</td>
</tr>
<tr>
<td>10.0</td>
<td>F</td>
<td>%f18</td>
<td></td>
<td></td>
<td>%f18</td>
</tr>
<tr>
<td>10.0</td>
<td>F</td>
<td>%f20</td>
<td></td>
<td></td>
<td>%f20</td>
</tr>
</tbody>
</table>

**Value**

- %f0: 20.0
- %f2: 10.0
- %f4: 10.0
- %f6: 10.0
- %f8: 10.0
- %f10: 10.0
- %f12: 10.0
- %f16: 10.0
- %f18: 10.0
- %f20: 10.0

**Operands**

- ADD: 1200, 80.0, %f14
- MULT: 1200, 80.0, %f14

**Actions**
- Instruction w retired
  - Instruction x executed

**ROB:**

<table>
<thead>
<tr>
<th>Value</th>
<th>Valid</th>
<th>Actual</th>
<th>OP1</th>
<th>OP2</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.0</td>
<td>T</td>
<td>%f0</td>
<td></td>
<td></td>
<td>v</td>
</tr>
<tr>
<td>10.0</td>
<td>T</td>
<td>%f2</td>
<td></td>
<td></td>
<td>%f2</td>
</tr>
<tr>
<td>20.0</td>
<td>T</td>
<td>%f4</td>
<td></td>
<td></td>
<td>%f4</td>
</tr>
<tr>
<td>40.0</td>
<td>T</td>
<td>%f6</td>
<td></td>
<td></td>
<td>%f6</td>
</tr>
<tr>
<td>80.0</td>
<td>T</td>
<td>%f8</td>
<td></td>
<td></td>
<td>%f8</td>
</tr>
<tr>
<td>160.0</td>
<td>T</td>
<td>%f10</td>
<td></td>
<td></td>
<td>%f10</td>
</tr>
<tr>
<td>320.0</td>
<td>T</td>
<td>%f12</td>
<td></td>
<td></td>
<td>%f12</td>
</tr>
<tr>
<td>30.0</td>
<td>F</td>
<td>%f16</td>
<td></td>
<td></td>
<td>%f16</td>
</tr>
<tr>
<td>10.0</td>
<td>F</td>
<td>%f18</td>
<td></td>
<td></td>
<td>%f18</td>
</tr>
<tr>
<td>10.0</td>
<td>F</td>
<td>%f20</td>
<td></td>
<td></td>
<td>%f20</td>
</tr>
</tbody>
</table>

**Value**

- %f0: 60.0
- %f2: 60.0
- %f4: 60.0
- %f6: 60.0
- %f8: 60.0
- %f10: 60.0
- %f12: 60.0
- %f16: 60.0
- %f18: 60.0
- %f20: 60.0

**Operands**

- ADD: 1240, 40.0, %f18
- MULT: 1240, 40.0, %f18
Execution Example: Cycle 6

- Instruction x & y retired
- Instruction z executed

<table>
<thead>
<tr>
<th>Value</th>
<th>Valid</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f0$</td>
<td>T</td>
<td>$f0$</td>
</tr>
<tr>
<td>$f2$</td>
<td>T</td>
<td>$f2$</td>
</tr>
<tr>
<td>$f4$</td>
<td>T</td>
<td>$f4$</td>
</tr>
<tr>
<td>$f6$</td>
<td>T</td>
<td>$f6$</td>
</tr>
<tr>
<td>$f8$</td>
<td>T</td>
<td>$f8$</td>
</tr>
<tr>
<td>$f10$</td>
<td>T</td>
<td>$f10$</td>
</tr>
<tr>
<td>$f12$</td>
<td>T</td>
<td>$f12$</td>
</tr>
<tr>
<td>$f14$</td>
<td>T</td>
<td>$f14$</td>
</tr>
<tr>
<td>$f16$</td>
<td>T</td>
<td>$f16$</td>
</tr>
<tr>
<td>$f18$</td>
<td>T</td>
<td>$f18$</td>
</tr>
<tr>
<td>$f20$</td>
<td>T</td>
<td>$f20$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Valid</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f0$</td>
<td>T</td>
<td>$f0$</td>
</tr>
<tr>
<td>$f2$</td>
<td>T</td>
<td>$f2$</td>
</tr>
<tr>
<td>$f4$</td>
<td>T</td>
<td>$f4$</td>
</tr>
<tr>
<td>$f6$</td>
<td>T</td>
<td>$f6$</td>
</tr>
<tr>
<td>$f8$</td>
<td>T</td>
<td>$f8$</td>
</tr>
<tr>
<td>$f10$</td>
<td>T</td>
<td>$f10$</td>
</tr>
<tr>
<td>$f12$</td>
<td>T</td>
<td>$f12$</td>
</tr>
<tr>
<td>$f14$</td>
<td>T</td>
<td>$f14$</td>
</tr>
<tr>
<td>$f16$</td>
<td>T</td>
<td>$f16$</td>
</tr>
<tr>
<td>$f18$</td>
<td>T</td>
<td>$f18$</td>
</tr>
<tr>
<td>$f20$</td>
<td>T</td>
<td>$f20$</td>
</tr>
</tbody>
</table>

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

Branch Prediction

Branch History Table (BHT)
- 512 state machines, indexed by low-order bits of instruction address
- Encode information about prior history of branch instructions
  - Small chance of two branch instructions aliasing
- Predict whether or not branch will be taken
  - 3 cycle penalty if mispredict

Interaction with BTAC
- BHT entries start in state No!
- When make transition from No? to Yes?, allocate entry in BTAC
- Deallocate when make transition from Yes? to No!
Branch Prediction Example

```java
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;
        ax = ABS(x);
        abs_sum += ax;
        prod *= x;
    }
    answer = abs_sum+prod;
}
```

#define ABS(x) x < 0 ? -x : x

MIPS Code

```mips
0x6c4: 8c620000 lw r2,0(r3)
0x6c8: 24840001 addiu r4,r4,10
0x6cc: 04410002 bgez r2,0x6d8
0x6d0: 00a20018 mult r5,r2
0x6d4: 00021023 subu r2,r0,r2
0x6d8: 00021012 mult r5
0x6dc: 00c23021 addu r6,r6,r2
0x6e0: 28820400 slti r2,r4,1024
0x6e4: 1440fff7 bne r2,r0,0x6c4
0x6e8: 24630004 addiu r3,r3,4
```

Some Interesting Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>R3000</th>
<th>PPC 604</th>
<th>Pentium</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PPPPPPP</strong></td>
<td>13.6</td>
<td>9.2</td>
<td>21.1</td>
</tr>
<tr>
<td><strong>RRRRRRRRR</strong></td>
<td>13.6</td>
<td>12.6</td>
<td>22.9</td>
</tr>
<tr>
<td><strong>N*N[PPPN]</strong></td>
<td>13.6</td>
<td>15.9</td>
<td>23.3</td>
</tr>
<tr>
<td><strong>N*P[PPPN]</strong></td>
<td>13.3</td>
<td>6.6</td>
<td>24.3</td>
</tr>
<tr>
<td><strong>N*N[PPNN]</strong></td>
<td>13.3</td>
<td>15.9</td>
<td>33.3</td>
</tr>
<tr>
<td><strong>N*P[PPNN]</strong></td>
<td>13.3</td>
<td>12.5</td>
<td>37.9</td>
</tr>
<tr>
<td><strong>N*P[PPPN]</strong></td>
<td>13.6</td>
<td>12.5</td>
<td>24.7</td>
</tr>
</tbody>
</table>

Loop Performance (FP)

Observations

- 604 has prediction rates 0%, 50%, and 100%
  - Expected 50% from N*N[PNPN]
  - Expected 25% from N*N[PPNN]
  - Loop so tight that speculate through single branch twice?
- 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”

Pentium II

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

R10000

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

Effect of Loop Unrolling

<table>
<thead>
<tr>
<th>Pattern</th>
<th>PPC State 1</th>
<th>PPC State 1X</th>
<th>PPC State 2X</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPPPPPPP</td>
<td>9.2</td>
<td>0</td>
<td>7.7</td>
</tr>
<tr>
<td>PRRRRRPR</td>
<td>12.8</td>
<td>3.4</td>
<td>11.3</td>
</tr>
<tr>
<td>N*N [PNPN]</td>
<td>15.8</td>
<td>6.6</td>
<td>13.8</td>
</tr>
<tr>
<td>N*P [PNPN]</td>
<td>15.9</td>
<td>6.7</td>
<td>14.7</td>
</tr>
<tr>
<td>N*N [PPNN]</td>
<td>12.5</td>
<td>3.3</td>
<td>11.3</td>
</tr>
<tr>
<td>N*P [PPNN]</td>
<td>12.5</td>
<td>3.3</td>
<td>14.3</td>
</tr>
</tbody>
</table>

Branch Prediction Algorithm

DEC Alpha 21264

Uses Every Trick in the Book
- 4-6 way superscalar
- Out of order execution with renaming
- Up to 80 instructions in process simultaneously
- Lots of cache & memory bandwidth

Another stressor in the life of a benchmarker
- Must look carefully at what compiler is doing
21264 Block Diagram

4 Integer ALUs
• Each can perform simple instructions
• 2 handle address calculations

Register Files
• 32 arch / 80 physical Int
• 32 arch / 72 physical FP
• Int registers duplicated
  - Extra cycle delay from write in one to read in other
  - Each has 6 read ports, 4 write ports
  - Attempt to issue consumer to producer side.

Microprocessor Report 10/28/96

21264 Pipeline

Very Deep Pipeline
• Can't do much in 2ns clock cycle!
  • 7 cycles for simple instruction
  • 9 cycles for load or store
  • 7 cycle penalty for mispredicted branch
    - Elaborate branch prediction logic
    - Claim 95% accuracy

Microprocessor Report 10/28/96

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

Microprocessor Report 12/30/96

Processor Comparisons

Microprocessor Report 12/30/96
**Challenges Ahead**

**Diminishing Returns on Cost vs. Performance**
- Superscalar processors require instruction level parallelism
- Many programs limited by sequential dependencies

**Finding New Sources of Parallelism**
- e.g., thread-level parallelism

**Getting Design Correct Difficult**
- Verification team larger than design team
- Devise tests for interactions between concurrent instructions
  - May be 80 executing at once

---

**New Era for Performance Optimization**

**Data Resources are Free and Fast**
- Plenty of computational units
- Most programs have poor utilization

**Unexpected Changes in Control Flow Expensive**
- Kill everything downstream when mispredict
- Even if will execute in near future where branches reconverge

**Think Parallel**
- Try to get lots of things going at once

**Not a Truly Parallel Machine**
- Bounded resources
- Access from limited code window

---

**What Next?**

- Instruction bandwidth
- Data bandwidth
- Dataflow limit

---

**Exceeding Dataflow limit**

- Value Prediction
- Speculation across control flow
- Predication