Superscalar Processing

CS 740
October 1, 1998

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
### 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>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i286</td>
<td>‘83</td>
<td>134K</td>
<td>8</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>i486</td>
<td></td>
<td>50</td>
<td>28 / 13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium</td>
<td>‘93</td>
<td>3.1M</td>
<td>66</td>
<td>78 / 64</td>
<td>4.3 / 3.0</td>
</tr>
<tr>
<td>Pentium</td>
<td></td>
<td>150</td>
<td>181 / 125</td>
<td>6.1 / 4.8</td>
<td>8.2 / 6.0</td>
</tr>
<tr>
<td>PentiumPro</td>
<td>‘95</td>
<td>5.5M</td>
<td>150</td>
<td>245 / 220</td>
<td>11.6 / 6.8</td>
</tr>
<tr>
<td>PentiumPro</td>
<td></td>
<td>200</td>
<td>320 / 283</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pentium II</td>
<td>‘97</td>
<td>7.5M</td>
<td>300</td>
<td></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</th>
<th>Spec95</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS R3000</td>
<td>‘88</td>
<td></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>
<td></td>
</tr>
<tr>
<td>MIPS R5000</td>
<td></td>
<td>3.6M</td>
<td>180</td>
<td>4.1 / 4.4</td>
<td></td>
</tr>
<tr>
<td>(Wean Hall SGI)</td>
<td></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</td>
<td>8.9 / 17.2</td>
</tr>
<tr>
<td>(Most Advanced MIPS)</td>
<td></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</td>
<td>11 / 17</td>
</tr>
<tr>
<td>(Fastest Available)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alpha 21264</td>
<td>‘97</td>
<td>15M</td>
<td>500</td>
<td>30 / 60</td>
<td></td>
</tr>
<tr>
<td>(Fastest Announced)</td>
<td></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>
x86 ISA Characteristics

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

<table>
<thead>
<tr>
<th>Generated</th>
<th>Used</th>
<th>Handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>ALU</td>
<td>EX–EX Forwarding</td>
</tr>
<tr>
<td>Load</td>
<td>ALU</td>
<td>EX–EX Forwarding</td>
</tr>
<tr>
<td>ALU</td>
<td>Store</td>
<td>EX–EX Forwarding</td>
</tr>
<tr>
<td>ALU</td>
<td>Eff. Address</td>
<td>(Stall) + EX–ID2 Forwarding</td>
</tr>
</tbody>
</table>


Control Hazards

Jump Instruction Processing

- Continue pipeline assuming branch not taken
- Resolve branch condition in EX stage
- Also speculatively fetch at target during EX stage
Control Hazards (Cont.)

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

(Microprocessor Report 10/28/92)
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

U-Pipe

Decode Control Word
Generate Memory Address

Access data cache or calculate ALU result

Write register result

V-Pipe
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

\[(\text{Single adder, data dependence}) \quad (\text{in order})\]

\[v: \text{addt } f2, f4, f10\]
\[w: \text{mult } f10, f6, f10\]
\[x: \text{addt } f10, f8, f12\]
\[y: \text{addt } f4, f6, f4\]
\[z: \text{addt } f4, f8, f10\]

Data Flow

Critical Path = 9 cycles
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: mult $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
PentiumPro Block Diagram
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
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
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
  \[ \geq 3 \text{ cycle penalty if mispredict} \]

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

“Reservation Stations”
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: \texttt{addq r1, r2, r3}
S2: \texttt{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}

...  
\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}

...  
\text{addq r8, r6, r7}
Write-after-Write (WAW) Dependences

Also known as an “output” dependence

Example:

\[
\begin{align*}
S1: & \quad \text{addq } r1, r2, r3 \\
S2: & \quad \text{addq } r4, r5, r3 \\
\quad \text{...} \\
\quad \text{addq } r3, r6, r7
\end{align*}
\]

How to optimize?

- rename dependent register (e.g., r3 in S2 -> r8)

\[
\begin{align*}
S1: & \quad \text{addq } r1, r2, r3 \\
S2: & \quad \text{addq } r4, r5, r8 \\
\quad \text{...} \\
\quad \text{addq } r8, r6, r7
\end{align*}
\]
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
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

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>Rename</th>
</tr>
</thead>
<tbody>
<tr>
<td>%f2</td>
<td>10.0</td>
</tr>
<tr>
<td>%f4</td>
<td>20.0</td>
</tr>
<tr>
<td>%f6</td>
<td>40.0</td>
</tr>
<tr>
<td>%f8</td>
<td>80.0</td>
</tr>
<tr>
<td>%f10</td>
<td>160.0</td>
</tr>
<tr>
<td>%f12</td>
<td>320.0</td>
</tr>
</tbody>
</table>

Value Renames

<table>
<thead>
<tr>
<th>%f0</th>
<th>%f2</th>
<th>%f4</th>
<th>%f6</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

ADD

MULT

Op1 Op2 Dest

Op1 Op2 Dest

Result Dest

Result Dest

v: addt %f2, %f4, %f10
w: mult %f10, %f6, %f10
x: addt %f10, %f8, %f12
y: addt %f4, %f6, %f4
z: addt %f4, %f8, %f10
Execution Example Cycle 1

Actions

- Instructions v & w issued
  - v target set to %f0
  - w target set to %f2

<table>
<thead>
<tr>
<th>Value</th>
<th>Rename</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f2</td>
<td>10.0</td>
</tr>
<tr>
<td>$f4</td>
<td>20.0</td>
</tr>
<tr>
<td>$f6</td>
<td>40.0</td>
</tr>
<tr>
<td>$f8</td>
<td>80.0</td>
</tr>
<tr>
<td>$f10</td>
<td>160.0</td>
</tr>
<tr>
<td>$f12</td>
<td>320.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Renames</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>%f0</td>
<td>--</td>
<td>$f10</td>
</tr>
<tr>
<td>%f2</td>
<td>--</td>
<td>$f10</td>
</tr>
<tr>
<td>%f4</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>%f6</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

V:  addt $f2, $f4, $f10
W:  mult $f10, $f6, $f10
X:  addt $f10, $f8, $f12
Y:  addt $f4, $f6, $f4
Z:  addt $f4, $f8, $f10

Op1     Op2     Dest
---      ---      ---
10.0     20.0     %f0
%f0      40.0     %f2

ADD

Result Dest
---      ---

MULT

Result Dest
---      ---
**Execution Example Cycle 2**

**Actions**

- **Instructions x & y issued**
  - x & y targets set to $f_6$ and $f_8$
- **Instruction v executed**

<table>
<thead>
<tr>
<th>Value</th>
<th>Rename</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_2$</td>
<td>10.0</td>
</tr>
<tr>
<td>$f_4$</td>
<td>20.0</td>
</tr>
<tr>
<td>$f_6$</td>
<td>40.0</td>
</tr>
<tr>
<td>$f_8$</td>
<td>80.0</td>
</tr>
<tr>
<td>$f_{10}$</td>
<td>160.0</td>
</tr>
<tr>
<td>$f_{12}$</td>
<td>320.0</td>
</tr>
</tbody>
</table>

**Value Renames**

<table>
<thead>
<tr>
<th>%f0</th>
<th>%f2</th>
<th>%f4</th>
<th>%f6</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.0</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

**ADD**

<table>
<thead>
<tr>
<th>Op1</th>
<th>Op2</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.0</td>
<td>%f0</td>
</tr>
</tbody>
</table>

**MULT**

<table>
<thead>
<tr>
<th>Op1</th>
<th>Op2</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.0</td>
<td>40.0</td>
<td>%f2</td>
</tr>
</tbody>
</table>

**Instructions**

- v: `addt $f_2, $f_4, $f_{10}`
- w: `mult $f_{10}, $f_6, $f_{10}`
- x: `addt $f_{10}, $f_8, $f_{12}`
- y: `addt $f_4, $f_6, $f_4`
- z: `addt $f_4, $f_8, $f_{10}`
• **Instruction v retired**
  – But doesn’t change $f_{10}$

• **Instruction w begins execution**
  – Moves through 2 stage pipeline

• **Instruction y executed**

• **Instruction z stalled**
  – Not enough reservation stations

---

**Value Rename**

<table>
<thead>
<tr>
<th>Value</th>
<th>Rename</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{2}$</td>
<td>10.0 $f_{2}$</td>
</tr>
<tr>
<td>$f_{4}$</td>
<td>20.0 $f_{6}$</td>
</tr>
<tr>
<td>$f_{6}$</td>
<td>40.0 $f_{6}$</td>
</tr>
<tr>
<td>$f_{8}$</td>
<td>80.0 $f_{8}$</td>
</tr>
<tr>
<td>$f_{10}$</td>
<td>160.0 $f_{2}$</td>
</tr>
<tr>
<td>$f_{12}$</td>
<td>320.0 $f_{4}$</td>
</tr>
</tbody>
</table>

**Value Renames**

<table>
<thead>
<tr>
<th>Value</th>
<th>Renames</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{0}$</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>$f_{2}$</td>
<td>--</td>
<td>$f_{10}$</td>
</tr>
<tr>
<td>$f_{4}$</td>
<td>--</td>
<td>$f_{12}$</td>
</tr>
<tr>
<td>$f_{6}$</td>
<td>60.0</td>
<td>$f_{4}$</td>
</tr>
</tbody>
</table>

---

**Cycle 3**

**Operators**

- **ADD**
  - **Op1**: $f_{2}$
  - **Op2**: 80.0
  - **Dest**: $f_{4}$

**Instructions**

- **v**: addt $f_{2}$, $f_{4}$, $f_{10}$
- **w**: mult $f_{10}$, $f_{6}$, $f_{10}$
- **x**: addt $f_{10}$, $f_{8}$, $f_{12}$
- **y**: addt $f_{4}$, $f_{6}$, $f_{4}$
- **z**: addt $f_{4}$, $f_{8}$, $f_{10}$
Execution Example Cycle 4

• Instruction w finishes execution
• Instruction y cannot be retired yet
• Instruction z issued
  – Assigned to %f0

<table>
<thead>
<tr>
<th>Value</th>
<th>Rename</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f2</td>
<td>10.0</td>
</tr>
<tr>
<td>$f4</td>
<td>20.0</td>
</tr>
<tr>
<td>$f6</td>
<td>40.0</td>
</tr>
<tr>
<td>$f8</td>
<td>80.0</td>
</tr>
<tr>
<td>$f10</td>
<td>160.0</td>
</tr>
<tr>
<td>$f12</td>
<td>320.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Renames</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>%f0</td>
<td>--</td>
<td>F</td>
</tr>
<tr>
<td>%f2</td>
<td>$f10</td>
<td>F</td>
</tr>
<tr>
<td>%f4</td>
<td>$f10</td>
<td>T</td>
</tr>
<tr>
<td>%f6</td>
<td>--</td>
<td>F</td>
</tr>
<tr>
<td>60</td>
<td>$f4</td>
<td>T</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Op1</th>
<th>Op2</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>80</td>
<td>%f0</td>
</tr>
<tr>
<td>120</td>
<td>80</td>
<td>%f4</td>
</tr>
</tbody>
</table>

ADD

<table>
<thead>
<tr>
<th>Result</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

MULT

<table>
<thead>
<tr>
<th>Result</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>120.0</td>
<td>%f2</td>
</tr>
</tbody>
</table>
Execution Example Cycle 5

- **Instruction w retired**
  - But does not change $f_{10}$
- **Instruction y cannot be retired yet**
- **Instruction x executed**

<table>
<thead>
<tr>
<th>Value</th>
<th>Rename</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{2}$</td>
<td>10.0</td>
</tr>
<tr>
<td>$f_{4}$</td>
<td>20.0</td>
</tr>
<tr>
<td>$f_{6}$</td>
<td>40.0</td>
</tr>
<tr>
<td>$f_{8}$</td>
<td>80.0</td>
</tr>
<tr>
<td>$f_{10}$</td>
<td>160.0</td>
</tr>
<tr>
<td>$f_{12}$</td>
<td>320.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Renames</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{0}$</td>
<td>--</td>
<td>$f_{10}$</td>
</tr>
<tr>
<td>$f_{2}$</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>$f_{4}$</td>
<td>200.0</td>
<td>$f_{12}$</td>
</tr>
<tr>
<td>$f_{6}$</td>
<td>60</td>
<td>$f_{4}$</td>
</tr>
</tbody>
</table>

- **ADD**
  - v: `addt $f_{2}, f_{4}, f_{10}`
  - w: `mult $f_{10}, f_{6}, f_{10}`
  - x: `addt $f_{10}, f_{8}, f_{12}`
  - y: `addt $f_{4}, f_{6}, f_{4}`
  - z: `addt $f_{4}, f_{8}, f_{10}`

- **MULT**
  - x: `addt $f_{10}, f_{8}, f_{12}`
  - y: `addt $f_{4}, f_{6}, f_{4}`
  - z: `addt $f_{4}, f_{8}, f_{10}`

<table>
<thead>
<tr>
<th>Op1</th>
<th>Op2</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>60.0</td>
<td>80.0</td>
<td>$f_{0}$</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>200.0</td>
<td>$f_{12}$</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Op1</th>
<th>Op2</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>
Execution Example Cycle 6

- **Instruction x & y retired**
  - Update $f_{12}$ and $f_{4}$
- **Instruction z executed**

### Values and Renames

<table>
<thead>
<tr>
<th>Value</th>
<th>Rename</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_{2}$</td>
<td>10.0</td>
</tr>
<tr>
<td>$f_{4}$</td>
<td>60.0</td>
</tr>
<tr>
<td>$f_{6}$</td>
<td>40.0</td>
</tr>
<tr>
<td>$f_{8}$</td>
<td>80.0</td>
</tr>
<tr>
<td>$f_{10}$</td>
<td>160.0</td>
</tr>
<tr>
<td>$f_{12}$</td>
<td>200.0</td>
</tr>
</tbody>
</table>

### Value Renames

<table>
<thead>
<tr>
<th>Value</th>
<th>Renames</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>%f0</td>
<td>140.0</td>
<td>$f_{10}$</td>
</tr>
<tr>
<td>%f2</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>%f4</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>%f6</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

### Operations

<table>
<thead>
<tr>
<th>Op1</th>
<th>Op2</th>
<th>Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

### Results

<table>
<thead>
<tr>
<th>z</th>
<th>140.0</th>
<th>%f0</th>
</tr>
</thead>
</table>

- **v:** `adtt $f_{2}, $f_{4}, $f_{10}`
- **w:** `mult $f_{10}, $f_{6}, $f_{10}`
- **x:** `adtt $f_{10}, $f_{8}, $f_{12}`
- **y:** `adtt $f_{4}, $f_{6}, $f_{4}`
- **z:** `adtt $f_{4}, $f_{8}, $f_{10}`
Execution Example Cycle 7

- Instruction z retired

<table>
<thead>
<tr>
<th>Value</th>
<th>Rename</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f2</td>
<td>10.0</td>
</tr>
<tr>
<td>$f4</td>
<td>60.0</td>
</tr>
<tr>
<td>$f6</td>
<td>40.0</td>
</tr>
<tr>
<td>$f8</td>
<td>80.0</td>
</tr>
<tr>
<td>$f10</td>
<td>140.0</td>
</tr>
<tr>
<td>$f12</td>
<td>320.0</td>
</tr>
</tbody>
</table>

V:      addt $f2, $f4, $f10  
W:      mult $f10, $f6, $f10  
X:      addt $f10, $f8, $f12  
Y:      addt $f4, $f6, $f4  
Z:      addt $f4, $f8, $f10  

<table>
<thead>
<tr>
<th>Value</th>
<th>Renames</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>%f0</td>
<td>--</td>
<td>F</td>
</tr>
<tr>
<td>%f2</td>
<td>--</td>
<td>F</td>
</tr>
<tr>
<td>%f4</td>
<td>--</td>
<td>F</td>
</tr>
<tr>
<td>%f6</td>
<td>--</td>
<td>F</td>
</tr>
</tbody>
</table>

Op1 | Op2 | Dest |
---  | ---  | ---  |
---  | ---  | ---  |
---  | ---  | ---  |

ADD  

Op1 | Op2 | Dest |
---  | ---  | ---  |
---  | ---  | ---  |
---  | ---  | ---  |

MULT 

Result | Dest | 
--- | --- |
--- | --- |
--- | --- |
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 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;
        ax = ABS(x);
        abs_sum += ax;
        prod *= x;
    }
    answer = abs_sum+prod;
}

• Compute sum of absolute values
• Compute product of original values

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

MIPS Code

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x6c4:</td>
<td>8c620000 lw r2,0(r3)</td>
</tr>
<tr>
<td>0x6c8:</td>
<td>24840001 addiu r4,r4,1</td>
</tr>
<tr>
<td>0x6cc:</td>
<td>04410002 bgez r2,0x6d8</td>
</tr>
<tr>
<td>0x6d0:</td>
<td>00a20018 mult r5,r2</td>
</tr>
<tr>
<td>0x6d4:</td>
<td>00021023 subu r2,r0,r2</td>
</tr>
<tr>
<td>0x6d8:</td>
<td>00002812 mflo r5</td>
</tr>
<tr>
<td>0x6dc:</td>
<td>00c23021 addu r6,r6,r2</td>
</tr>
<tr>
<td>0x6e0:</td>
<td>28820400 slti r2,r4,1024</td>
</tr>
<tr>
<td>0x6e4:</td>
<td>1440fff7 bne r2,r0,0x6c4</td>
</tr>
<tr>
<td>0x6e8:</td>
<td>24630004 addiu r3,r3,4</td>
</tr>
</tbody>
</table>
Some Interesting Patterns

PPPPPPPPPP
+1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 +1 ...  
  • Should give perfect prediction

RRRRRRRRRR
-1 -1 +1 +1 +1 +1 -1 -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[PNPN]
-1 -1 -1 -1 -1 -1 -1 +1 -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 +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 +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 +1 +1 -1 +1 +1 -1 -1 -1 ...  

## 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”

<table>
<thead>
<tr>
<th>Pattern</th>
<th>R3000 Cycles</th>
<th>Penalty</th>
<th>PPC 604 Cycles</th>
<th>Penalty</th>
<th>Pentium Cycles</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPPPPPPP</td>
<td>13.6</td>
<td>0</td>
<td>9.2</td>
<td>0</td>
<td>21.1</td>
<td>0</td>
</tr>
<tr>
<td>RRRRRRRR</td>
<td>13.6</td>
<td>0</td>
<td>12.6</td>
<td>3.4</td>
<td>22.9</td>
<td>1.8</td>
</tr>
<tr>
<td>N*N[PNPN]</td>
<td>13.6</td>
<td>0</td>
<td>15.8</td>
<td>6.6</td>
<td>23.3</td>
<td>2.2</td>
</tr>
<tr>
<td>N*P[PNPN]</td>
<td>13.3</td>
<td>-0.3</td>
<td>15.9</td>
<td>6.7</td>
<td>24.3</td>
<td>3.2</td>
</tr>
<tr>
<td>N*N[PPNN]</td>
<td>13.3</td>
<td>-0.3</td>
<td>12.5</td>
<td>3.3</td>
<td>23.9</td>
<td>2.8</td>
</tr>
<tr>
<td>N*P[PPNN]</td>
<td>13.6</td>
<td>0</td>
<td>12.5</td>
<td>3.3</td>
<td>24.7</td>
<td>3.6</td>
</tr>
</tbody>
</table>
Loop 1 Surprises

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
  
  \[
  a = Fexpr \\
  \text{temp} = Texpr \\
  \text{CMOV}(a, \text{temp}, \text{Cond})
  \]
- Only valid if $Texpr \ & \ Fexpr$ can’t cause error

<table>
<thead>
<tr>
<th>Pattern</th>
<th>R10000 Cycles</th>
<th>Penalty</th>
<th>Pentium II Cycles</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPPPPPPP</td>
<td>3.5</td>
<td>0</td>
<td>11.9</td>
<td>0</td>
</tr>
<tr>
<td>RRRRRRRR</td>
<td>3.5</td>
<td>0</td>
<td>19</td>
<td>7.1</td>
</tr>
<tr>
<td>N*N [PNPN]</td>
<td>3.5</td>
<td>0</td>
<td>12.5</td>
<td>0.6</td>
</tr>
<tr>
<td>N*P [PNPN]</td>
<td>3.5</td>
<td>0</td>
<td>13</td>
<td>1.1</td>
</tr>
<tr>
<td>N*N [PPNN]</td>
<td>3.5</td>
<td>0</td>
<td>12.4</td>
<td>0.5</td>
</tr>
<tr>
<td>N*P [PPNN]</td>
<td>3.5</td>
<td>0</td>
<td>12.2</td>
<td>0.3</td>
</tr>
</tbody>
</table>
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

Microprocessor Report March 27, 1995
Effect of Loop Unrolling

### Observations

- [PNPN] yields PPPP ... for one branch, NNNN ... for the other
- [PPNN] yields PNPN ... for both branches
  - 50% accuracy if start in state No?
  - 25% accuracy if start in state No!

**Another stressor in the life of a benchmark**

- Must look carefully at what compiler is doing
MIPS R10000

(See attached handouts.)

More info available at:
  • http://www.sgi.com/MIPS/products/r10k
DEC Alpha 21264

Fastest Announced Processor
- Spec95: 30 Int 60 FP
- 500 MHz, 15M transistors, 60 Watts

Fastest Existing Processor is Alpha 21164
- Spec95: 12.6 Int 18.3 FP
- 500 MHz, 9.2M transistors, 25 Watts

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
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
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 predication logic
  - Claim 95% accuracy
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
## Processor Comparisons

### Processors for Workstations and Servers

<table>
<thead>
<tr>
<th>Processor</th>
<th>Max Clock Speed</th>
<th>Cache Size</th>
<th>Supply Voltage</th>
<th>Max Power</th>
<th>Transistor Count</th>
<th>IC Process</th>
<th>Die Size</th>
<th>Est Mfg Cost*</th>
<th>SPEC95b int/fp</th>
<th>List Price</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital 21164</td>
<td>500 MHz</td>
<td>8K/8K/96K</td>
<td>2.0 V</td>
<td>25 W</td>
<td>9.3 million</td>
<td>0.35µ 4M</td>
<td>209 mm²</td>
<td>$150</td>
<td>12.6/18.3</td>
<td>$1,450</td>
<td>now</td>
</tr>
<tr>
<td>Digital 21264</td>
<td>&gt;500 MHz</td>
<td>64K/64K</td>
<td>2.0 V</td>
<td>60 W</td>
<td>15 million</td>
<td>0.35µ 6M</td>
<td>300 mm²</td>
<td>$300</td>
<td>30/60</td>
<td>N.D.</td>
<td>4Q97</td>
</tr>
<tr>
<td>Fuj. TurboSparc</td>
<td>170 MHz</td>
<td>16K/16K</td>
<td>3.3 V</td>
<td>9 W</td>
<td>3.0 million</td>
<td>0.35µ 4M</td>
<td>132 mm²</td>
<td>$50</td>
<td>3.5/3.0</td>
<td>$499</td>
<td>now</td>
</tr>
<tr>
<td>HP PA-7300LC</td>
<td>160 MHz</td>
<td>64K/64K</td>
<td>3.3 V</td>
<td>15 W</td>
<td>9.2 million</td>
<td>0.5µ 4M</td>
<td>259 mm²</td>
<td>$95</td>
<td>5.5/7.3</td>
<td>not sold</td>
<td>now</td>
</tr>
<tr>
<td>HP PA-8000</td>
<td>180 MHz</td>
<td>none</td>
<td>3.3 V</td>
<td>&gt;40 W</td>
<td>3.9 million</td>
<td>0.5µ 4M</td>
<td>345 mm²</td>
<td>$290</td>
<td>10.8/18.3</td>
<td>not sold</td>
<td>now</td>
</tr>
<tr>
<td>IBM P2SC</td>
<td>135 MHz</td>
<td>32K/128K</td>
<td>2.5 V</td>
<td>30 W</td>
<td>15 million</td>
<td>0.29µ 4M</td>
<td>335 mm²</td>
<td>$375</td>
<td>5.5/14.5</td>
<td>not sold</td>
<td>now</td>
</tr>
<tr>
<td>MIPS R5000</td>
<td>180 MHz</td>
<td>32K/32K</td>
<td>3.3 V</td>
<td>10 W</td>
<td>3.6 million</td>
<td>0.35µ 3M</td>
<td>84 mm²</td>
<td>$25</td>
<td>4.0/3.7</td>
<td>$365</td>
<td>now</td>
</tr>
<tr>
<td>MIPS R7000</td>
<td>300 MHz</td>
<td>288K(1)</td>
<td>3.3 V</td>
<td>13 W</td>
<td>N.D.</td>
<td>0.25µ 4M</td>
<td>80 mm²</td>
<td>$35</td>
<td>10/10</td>
<td>N.D.</td>
<td>2H97</td>
</tr>
<tr>
<td>MIPS R10000</td>
<td>200 MHz</td>
<td>32K/32K</td>
<td>3.3 V</td>
<td>30 W</td>
<td>5.9 million</td>
<td>0.35µ 4M</td>
<td>298 mm²</td>
<td>$160</td>
<td>8.9/17.2</td>
<td>$3,000</td>
<td>now</td>
</tr>
<tr>
<td>Sun UltraSparc-2</td>
<td>250 MHz</td>
<td>16K/16K</td>
<td>2.5 V</td>
<td>20 W</td>
<td>3.8 million</td>
<td>0.29µ 5M</td>
<td>149 mm²</td>
<td>$90</td>
<td>8.5/15</td>
<td>$1,995</td>
<td>limited</td>
</tr>
</tbody>
</table>

### Processors for PCS and Workstations

<table>
<thead>
<tr>
<th>Processor</th>
<th>Max Clock Speed</th>
<th>Cache Size</th>
<th>Supply Voltage</th>
<th>Max Power</th>
<th>Transistor Count</th>
<th>IC Process</th>
<th>Die Size</th>
<th>Est Mfg Cost*</th>
<th>SPEC95b int/fp</th>
<th>List Price</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential x704</td>
<td>553 MHz</td>
<td>2K/2K/32K</td>
<td>3.6 V</td>
<td>85 W</td>
<td>2.7 million</td>
<td>0.5µ 5M(3)</td>
<td>150 mm²</td>
<td>$90</td>
<td>12/10</td>
<td>$1,000*</td>
<td>2Q97</td>
</tr>
<tr>
<td>PowerPC 603e</td>
<td>240 MHz</td>
<td>16K/16K</td>
<td>2.5 V</td>
<td>6 W</td>
<td>2.6 million</td>
<td>0.35µ 4M</td>
<td>79 mm²</td>
<td>$30</td>
<td>5.5/4*</td>
<td>$408</td>
<td>now</td>
</tr>
<tr>
<td>PowerPC 604e</td>
<td>225 MHz</td>
<td>32K/32K</td>
<td>2.5 V</td>
<td>24 W</td>
<td>5.1 million</td>
<td>0.35µ 4M</td>
<td>148 mm²</td>
<td>$60</td>
<td>8/7*</td>
<td>$533</td>
<td>now</td>
</tr>
<tr>
<td>Intel Pentium</td>
<td>200 MHz</td>
<td>8K/8K</td>
<td>3.3 V</td>
<td>17 W</td>
<td>3.3 million</td>
<td>0.35µ 4M(3)</td>
<td>90 mm²</td>
<td>$40</td>
<td>5.5/2.9</td>
<td>$509</td>
<td>now</td>
</tr>
<tr>
<td>Intel P55C</td>
<td>200 MHz</td>
<td>16K/16K</td>
<td>2.8 V</td>
<td>16 W</td>
<td>4.5 million</td>
<td>0.28µ 4M</td>
<td>140 mm²</td>
<td>$50</td>
<td>6/3*</td>
<td>N.D.</td>
<td>1Q97</td>
</tr>
<tr>
<td>Intel PPro</td>
<td>200 MHz</td>
<td>8K/8K</td>
<td>3.3 V</td>
<td>35 W†</td>
<td>5.5 million</td>
<td>0.35µ 4M(3)</td>
<td>196 mm²</td>
<td>$145†</td>
<td>8.2/6.0†</td>
<td>$525†</td>
<td>now</td>
</tr>
<tr>
<td>Intel Klamath</td>
<td>266 MHz*</td>
<td>16K/16K</td>
<td>2.8 V*</td>
<td>N.D.</td>
<td>7.5 million</td>
<td>0.28µ 4M</td>
<td>203 mm²</td>
<td>$80</td>
<td>11/7*</td>
<td>N.D.</td>
<td>2Q97*</td>
</tr>
</tbody>
</table>

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