Computer Architecture
ELE 475 / COS 475
Slide Deck 7: VLIW

David Wentzlaff
Department of Electrical Engineering
Princeton University
Superscalar Control Logic Scaling

- Each issued instruction must somehow check against $W*L$ instructions, i.e.,
growth in hardware $\propto W*(W*L)$
- For in-order machines, $L$ is related to pipeline latencies and check is done during issue (scoreboard)
- For out-of-order machines, $L$ also includes time spent in IQ, SB, and check is done by broadcasting tags to waiting instructions at completion
- As $W$ increases, larger instruction window is needed to find enough parallelism to keep machine busy => greater $L$

$=>$ Out-of-order control logic grows faster than $W^2$ ($\sim W^3$)
Out-of-Order Control Complexity: MIPS R10000

[A. Ahi et al., MIPS R10000 Superscalar Microprocessor, Hot Chips, 1995]
Image Credit: MIPS Technologies Inc. / Silicon Graphics Computer Systems
Out-of-Order Control Complexity: MIPS R10000

[A. Ahi et al., MIPS R10000 Superscalar Microprocessor, Hot Chips, 1995]
Image Credit: MIPS Technologies Inc. / Silicon Graphics Computer Systems
Sequential ISA Bottleneck

Sequential source code

```
a = foo(b);
for (i=0, i<
```

Superscalar compiler

Find independent operations
Sequential ISA Bottleneck

Sequential source code

```
a = foo(b);
for (i=0, i<
```

Superscalar compiler

Find independent operations

Schedule operations
Sequential ISA Bottleneck

Sequential source code

\[ a = \text{foo}(b); \quad \text{for } (i=0, \ i< \]
Sequential ISA Bottleneck

Sequential source code
a = foo(b); for (i=0, i<

Superscalar compiler
Find independent operations Schedule operations

Superscalar processor
Check instruction dependencies Schedule execution

Sequential machine code
VLIW: Very Long Instruction Word

- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
  - Parallelism within an instruction => no cross-operation RAW check
  - No data use before data ready => no data interlocks
VLIW Equals (EQ) Scheduling Model

- Each operation takes exactly specified latency
- Efficient register usage (Effectively more registers)
- No need for register renaming or buffering
  - Bypass from functional unit output to inputs
  - Register writes whenever functional unit completes
- Compiler depends on not having registers visible early
VLIW Less-Than-or-Equals (LEQ) Scheduling Model

• Each operation may take less than or equal to its specified latency
  – Destination can be written any time after instruction issue
  – Dependent instruction still needs to be scheduled after instruction latency

• Precise interrupts simplified

• Binary compatibility preserved when latencies are reduced
Blackboard Example Equals Model

{MUL R1, R2, R3; ADD R7, R8, R9}
{AND R10, R1, R5}

X0
Y0 Y1 Y2 W
M0 M1
Extra Blackboard Example VLIW Code Sequence EQ Scheduling

{MUL R1, R3, R4; ADD R7, R8, R9}
{ADD R9, R1, R10}
{ADD; SUB}
{NOP}
{LW R17, R1}
Early VLIW Machines

• FPS AP120B (1976)
  – scientific attached array processor
  – first commercial wide instruction machine
  – hand-coded vector math libraries using software pipelining and loop unrolling

• Multiflow Trace (1987)
  – commercialization of ideas from Fisher’s Yale group including “trace scheduling”
  – available in configurations with 7, 14, or 28 operations/instruction
  – 28 operations packed into a 1024-bit instruction word

• Cydrome Cydra-5 (1987)
  – 7 operations encoded in 256-bit instruction word
  – rotating register file
VLIW Compiler Responsibilities

• Schedule operations to maximize parallel execution

• Guarantees intra-instruction parallelism

• Schedule to avoid data hazards (no interlocks)
  – Typically separates operations with explicit NOPs
for (i=0; i<N; i++)

loop: lw F1, 0(R1)
addiu R1, R1, 4
add.s F2, F0, F1
sw F2, 0(R2)
addiu R2, R2, 4
bne R1, R3, loop

Loop Execution

Compile

Schedule
for (i=0; i<N; i++)

Compile

loop:
  lw F1, 0(R1)
  addiu R1, R1, 4
  add.s F2, F0, F1
  sw F2, 0(R2)
  addiu R2, R2, 4
  bne R1, R3, loop

Schedule
for (i=0; i<N; i++)

loop:  lw F1, 0(R1)
       addiu R1, R1, 4
       add.s F2, F0, F1
       sw F2, 0(R2)
       addiu R2, R2, 4
       bne R1, R3, loop
for (i=0; i<N; i++)

Compile

loop: lw F1, 0(R1)
addiu R1, R1, 4
add.s F2, F0, F1
sw F2, 0(R2)
addiu R2, R2, 4
bne R1, R3, loop

Schedule

Loop Execution

Int1  Int 2  M1  M2  FP+  FPx

add R1    lw

add.s

sw

add R2
Loop Execution

for (i=0; i<N; i++)

Compile

loop:
    lw F1, 0(R1)
    addiu R1, R1, 4
    add.s F2, F0, F1
    sw F2, 0(R2)
    addiu R2, R2, 4
    bne R1, R3, loop

Schedule

How many FP ops/cycle?
for (i=0; i<N; i++)

loop:  lw F1, 0(R1)
        addiu R1, R1, 4
        add.s F2, F0, F1
        sw F2, 0(R2)
        addiu R2, R2, 4
        bne R1, R3, loop

How many FP ops/cycle?
1 add.s / 8 cycles = 0.125
Loop Unrolling

\[
\text{for (i=0; i<N; i++)}
\quad \text{B[i] = A[i] + C;}
\]

Unroll inner loop to perform 4 iterations at once

\[
\text{for (i=0; i<N; i+=4)}
\quad \{
\quad \quad \text{B[i] = A[i] + C;}
\quad \quad \text{B[i+1] = A[i+1] + C;}
\quad \quad \text{B[i+2] = A[i+2] + C;}
\quad \quad \text{B[i+3] = A[i+3] + C;}
\quad \}
\]
Loop Unrolling

for (i=0; i<N; i++)

Unroll inner loop to perform 4 iterations at once

for (i=0; i<N; i+=4)
{
}

Need to handle values of N that are not multiples of unrolling factor with final cleanup loop
Scheduling Loop Unrolled Code

Unroll 4 ways

<table>
<thead>
<tr>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

loop:

lw F1, 0(r1)
lw F2, 4(r1)
lw F3, 8(r1)
lw F4, 12(r1)
addiu R1, R1, 16
add.s F5, F0, F1
add.s F6, F0, F2
add.s F7, F0, F3
add.s F8, F0, F4
sw F5, 0(R2)
sw F6, 4(R2)
sw F7, 8(R2)
sw F8, 12(R2)
addiu R2, R2, 16
bne R1, R3, loop

Schedule
Scheduling Loop Unrolled Code

**Unroll 4 ways**

```assembly
loop:
lw  F1, 0(r1)
lw  F2, 4(r1)
lw  F3, 8(r1)
lw  F4, 12(r1)
addiu R1, R1, 16
add.s F5, F0, F1
add.s F6, F0, F2
add.s F7, F0, F3
add.s F8, F0, F4
sw  F5, 0(R2)
sw  F6, 4(R2)
sw  F7, 8(R2)
sw  F8, 12(R2)
addiu R2, R2, 16
bne R1, R3, loop
```

---

### Schedule

![Schedule](image_url)
Scheduling Loop Unrolled Code

Unroll 4 ways

loop: lw F1, 0(r1)
lw F2, 4(r1)
lw F3, 8(r1)
lw F4, 12(r1)
addiu R1, R1, 16
add.s F5, F0, F1
add.s F6, F0, F2
add.s F7, F0, F3
add.s F8, F0, F4
sw F5, 0(R2)
sw F6, 4(R2)
sw F7, 8(R2)
sw F8, 12(R2)
addiu R2, R2, 16
bne R1, R3, loop
**Scheduling Loop Unrolled Code**

### Unroll 4 ways

| loop: lw F1, 0(r1) lw F2, 4(r1) lw F3, 8(r1) lw F4, 12(r1) addiu R1, R1, 16 add.s F5, F0, F1 add.s F6, F0, F2 add.s F7, F0, F3 add.s F8, F0, F4 sw F5, 0(R2) sw F6, 4(R2) sw F7, 8(R2) sw F8, 12(R2) addiu R2, R2, 16 bne R1, R3, loop |
|---|---|---|---|---|---|
| Int1 | Int 2 | M1 | M2 | FP+ | FPx |
| loop: | | | | | |

### Schedule

<table>
<thead>
<tr>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw F1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add R1</td>
<td>lw F4</td>
<td>add.s F5</td>
<td></td>
<td>add.s F6</td>
<td></td>
</tr>
<tr>
<td>add R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How many FLOPS/cycle?

27
Scheduling Loop Unrolled Code

Unroll 4 ways

loop:  lw F1, 0(r1)
       lw F2, 4(r1)
       lw F3, 8(r1)
       lw F4, 12(r1)
       addiu R1, R1, 16
       add.s F5, F0, F1
       add.s F6, F0, F2
       add.s F7, F0, F3
       add.s F8, F0, F4
       sw F5, 0(R2)
       sw F6, 4(R2)
       sw F7, 8(R2)
       sw F8, 12(R2)
       addiu R2, R2, 16
       bne R1, R3, loop

How many FLOPS/cycle?

4 add.s / 11 cycles = 0.36
Software Pipelining

Unroll 4 ways first

loop: lw F1, 0(R1)
lw F2, 4(R1)
lw F3, 8(R1)
lw F4, 12(R1)
addiu R1, R1, 16
add.s F5, F0, F1
add.s F6, F0, F2
add.s F7, F0, F3
add.s F8, F0, F4
sw F5, 0(R2)
sw F6, 4(R2)
sw F7, 8(R2)
sw F8, 12(R2)
addiu R2, R2, 16
bne R1, R3, loop
Software Pipelining

Unroll 4 ways first

loop:  
  lw F1, 0(R1)  
  lw F2, 4(R1)  
  lw F3, 8(R1)  
  lw F4, 12(R1)  
  addiu R1, R1, 16  
  add.s F5, F0, F1  
  add.s F6, F0, F2  
  add.s F7, F0, F3  
  add.s F8, F0, F4  
  sw F5, 0(R2)  
  sw F6, 4(R2)  
  sw F7, 8(R2)  
  sw F8, 12(R2)  
  addiu R2, R2, 16  
  bne R1, R3, loop

Int1 Int2 M1 M2 FP+ FPx

lw F1 lw F2
lw F3
lw F4
add R1 lw F4
add.s F5
add.s F6
add.s F7
add.s F8
sw F5 sw F6
sw F7
sw F8
add R2 sw F7
bne sw F8

### Software Pipelining

**Unroll 4 ways first**

<table>
<thead>
<tr>
<th></th>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>loop:</strong> lw F1, 0(R1)</td>
<td></td>
<td></td>
<td>lw F1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lw F2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lw F3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lw F4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>addiu R1, R1, 16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>add.s F5, F0, F1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>add.s F6, F0, F2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>add.s F7, F0, F3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>add.s F8, F0, F4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sw F5, 0(R2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sw F6, 4(R2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sw F7, 8(R2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>sw F8, 12(R2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu R2, R2, 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne R1, R3, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add R1</td>
<td>lw F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>add R1</td>
<td>lw F1</td>
<td>add.s F5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lw F2</td>
<td>add.s F6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lw F3</td>
<td>add.s F7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add R1</td>
<td></td>
<td>lw F4</td>
<td>add.s F8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw F5</td>
<td>add.s F5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw F6</td>
<td>add.s F6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add R2</td>
<td></td>
<td>sw F7</td>
<td>add.s F7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>sw F8</td>
<td>add.s F8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw F5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add R2</td>
<td>sw F7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>sw F8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

31
Software Pipelining

Unroll 4 ways first

<table>
<thead>
<tr>
<th></th>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop: lw F1, 0(R1)</td>
<td></td>
<td></td>
<td>lw F1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F2, 4(R1)</td>
<td></td>
<td></td>
<td>lw F2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F3, 8(R1)</td>
<td></td>
<td></td>
<td>lw F3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F4, 12(R1)</td>
<td></td>
<td></td>
<td>add R1</td>
<td>lw F4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addiu R1, R1, 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add.s F5, F0, F1</td>
<td></td>
<td></td>
<td>lw F1</td>
<td>add.s F5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add.s F6, F0, F2</td>
<td></td>
<td></td>
<td>lw F2</td>
<td>add.s F6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add.s F7, F0, F3</td>
<td></td>
<td></td>
<td>lw F3</td>
<td>add.s F7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add.s F8, F0, F4</td>
<td></td>
<td></td>
<td>add R1</td>
<td>lw F4</td>
<td>add.s F8</td>
<td></td>
</tr>
<tr>
<td>sw F5, 0(R2)</td>
<td></td>
<td></td>
<td>lw F1</td>
<td>sw F5</td>
<td>add.s F5</td>
<td></td>
</tr>
<tr>
<td>sw F6, 4(R2)</td>
<td></td>
<td></td>
<td>lw F2</td>
<td>sw F6</td>
<td>add.s F6</td>
<td></td>
</tr>
<tr>
<td>sw F7, 8(R2)</td>
<td></td>
<td></td>
<td>add R2</td>
<td>lw F3</td>
<td>sw F7</td>
<td>add.s F7</td>
</tr>
<tr>
<td>sw F8, 12(R2)</td>
<td></td>
<td></td>
<td>add R1</td>
<td>bne</td>
<td>lw F4</td>
<td>sw F8</td>
</tr>
<tr>
<td>addiu R2, R2, 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne R1, R3, loop</td>
<td></td>
<td></td>
<td>sw F5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Software Pipelining

Unroll 4 ways first

loop: lw F1, 0(R1)
lw F2, 4(R1)
lw F3, 8(R1)
lw F4, 12(R1)
addiu R1, R1, 16
add.s F5, F0, F1
add.s F6, F0, F2
add.s F7, F0, F3
add.s F8, F0, F4
sw F5, 0(R2)
sw F6, 4(R2)
sw F7, 8(R2)
sw F8, 12(R2)
addiu R2, R2, 16
bne R1, R3, loop

prolog

iterate

loop:

epilog

<table>
<thead>
<tr>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>lw F1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lw F2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lw F3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lw F4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add R1</td>
<td>lw F4</td>
<td></td>
<td></td>
<td>add.s F5</td>
<td></td>
</tr>
<tr>
<td>add R1</td>
<td>lw F4</td>
<td></td>
<td></td>
<td>add.s F6</td>
<td></td>
</tr>
<tr>
<td>add R1</td>
<td>lw F4</td>
<td></td>
<td></td>
<td>add.s F7</td>
<td></td>
</tr>
<tr>
<td>add R1</td>
<td>lw F4</td>
<td>sw F5</td>
<td>add.s F5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add R2</td>
<td>lw F3</td>
<td>sw F7</td>
<td>add.s F7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add R1</td>
<td>bne</td>
<td>lw F4</td>
<td>sw F8</td>
<td>add.s F8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw F5</td>
<td></td>
<td>add.s F5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw F6</td>
<td></td>
<td>add.s F6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw F7</td>
<td></td>
<td>add.s F7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw F8</td>
<td></td>
<td>add.s F8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw F5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Software Pipelining

### Unroll 4 ways first

**loop:**
- `lw F1, 0(R1)`
- `lw F2, 4(R1)`
- `lw F3, 8(R1)`
- `lw F4, 12(R1)`
- `addiu R1, R1, 16`
- `add.s F5, F0, F1`
- `add.s F6, F0, F2`
- `add.s F7, F0, F3`
- `add.s F8, F0, F4`
- `sw F5, 0(R2)`
- `sw F6, 4(R2)`
- `sw F7, 8(R2)`
- `sw F8, 12(R2)`
- `addiu R2, R2, 16`
- `bne R1, R3, loop`

### How many FLOPS/cycle?

<table>
<thead>
<tr>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw F1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add R1</td>
<td>lw F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add R2</td>
<td>lw F3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add R1</td>
<td>lw F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw F5</td>
<td>add.s F5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw F6</td>
<td>add.s F6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw F7</td>
<td>add.s F7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw F8</td>
<td>add.s F8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add R1</td>
<td>lw F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw F6</td>
<td>add.s F6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add R2</td>
<td>lw F3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw F7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw F7</td>
<td>add.s F7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td>lw F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw F5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**prolog**

**iterate**

**epilog**
Software Pipelining

Unroll 4 ways first

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Instruction</th>
<th>Instruction</th>
<th>Instruction</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw F1, 0(R1)</td>
<td>lw F1</td>
<td>lw F1</td>
<td>lw F1</td>
<td>lw F1</td>
</tr>
<tr>
<td>lw F2, 4(R1)</td>
<td>lw F2</td>
<td>lw F2</td>
<td>lw F2</td>
<td>lw F2</td>
</tr>
<tr>
<td>lw F3, 8(R1)</td>
<td>lw F3</td>
<td>lw F3</td>
<td>lw F3</td>
<td>lw F3</td>
</tr>
<tr>
<td>lw F4, 12(R1)</td>
<td>lw F4</td>
<td>lw F4</td>
<td>lw F4</td>
<td>lw F4</td>
</tr>
<tr>
<td>addiu R1, R1, 16</td>
<td>add R1</td>
<td>add R1</td>
<td>add R1</td>
<td>add R1</td>
</tr>
<tr>
<td>add.s F5, F0, F1</td>
<td>add.s F5</td>
<td>add.s F5</td>
<td>add.s F5</td>
<td>add.s F5</td>
</tr>
<tr>
<td>add.s F6, F0, F2</td>
<td>add.s F6</td>
<td>add.s F6</td>
<td>add.s F6</td>
<td>add.s F6</td>
</tr>
<tr>
<td>add.s F7, F0, F3</td>
<td>add.s F7</td>
<td>add.s F7</td>
<td>add.s F7</td>
<td>add.s F7</td>
</tr>
<tr>
<td>add.s F8, F0, F4</td>
<td>add.s F8</td>
<td>add.s F8</td>
<td>add.s F8</td>
<td>add.s F8</td>
</tr>
<tr>
<td>sw F5, 0(R2)</td>
<td>sw F5</td>
<td>sw F5</td>
<td>sw F5</td>
<td>sw F5</td>
</tr>
<tr>
<td>sw F6, 4(R2)</td>
<td>sw F6</td>
<td>sw F6</td>
<td>sw F6</td>
<td>sw F6</td>
</tr>
<tr>
<td>sw F7, 8(R2)</td>
<td>sw F7</td>
<td>sw F7</td>
<td>sw F7</td>
<td>sw F7</td>
</tr>
<tr>
<td>sw F8, 12(R2)</td>
<td>sw F8</td>
<td>sw F8</td>
<td>sw F8</td>
<td>sw F8</td>
</tr>
<tr>
<td>addiu R2, R2, 16</td>
<td>add R2</td>
<td>add R2</td>
<td>add R2</td>
<td>add R2</td>
</tr>
<tr>
<td>bne R1, R3, loop</td>
<td>bne</td>
<td>bne</td>
<td>bne</td>
<td>bne</td>
</tr>
</tbody>
</table>

How many FLOPS/cycle?

4 add.s / 4 cycles = 1
Software Pipelining vs. Loop Unrolling

Software pipelining pays startup/wind-down costs only once per loop, not once per iteration.
What if there are no loops?

- Branches limit basic block size in control-flow intensive irregular code
- Difficult to find ILP in individual basic blocks

Basic block
- Single entry
- Single exit
Trace Scheduling [Fisher, Ellis]

- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Use *profiling feedback* or compiler heuristics to find common branch paths
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace
Trace Scheduling \cite{Fisher,Ellis}

- Pick string of basic blocks, a \textit{trace}, that represents most frequent branch path
- Use \textit{profiling feedback} or compiler heuristics to find common branch paths
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace
Problems with “Classic” VLIW

• Object-code compatibility
  – have to recompile all code for every machine, even for two machines in same generation

• Object code size
  – instruction padding wastes instruction memory/cache
  – loop unrolling/software pipelining replicates code

• Scheduling variable latency memory operations
  – caches and/or memory bank conflicts impose statically unpredictable variability

• Knowing branch probabilities
  – Profiling requires an significant extra step in build process

• Scheduling for statically unpredictable branches
  – optimal schedule varies with branch path

• Precise Interrupts can be challenging
  – Does fault in one portion of bundle fault whole bundle?
  – EQ Model has problem with single step, etc.
VLIW Instruction Encoding

- Schemes to reduce effect of unused fields
  - Compressed format in memory, expand on I-cache refill
    - used in Multiflow Trace
    - introduces instruction addressing challenge
  - Mark parallel groups
    - used in TMS320C6x DSPs, Intel IA-64
  - Provide a single-op VLIW instruction
    - Cydra-5 UniOp instructions
Predication

Problem: Mispredicted branches limit ILP
Solution: Eliminate hard to predict branches with predicated execution

Predication helps with small branch regions and/or branches that are hard to predict by turning control flow into data flow

Most basic form of predication: conditional moves

• `movz rd, rs, rt if ( R[rt] == 0 ) then R[rd] <- R[rs]`
• `movn rd, rs, rt if ( R[rt] != 0 ) then R[rd] <- R[rs]`

```
if (a<b)     slt R1, R2, R3     slt R1, R2, R3
  x=a       beq R1, R0, L1     movz R4, R2, R1
else         move R4, R2      movn R4, R3, R1
  x=b         j L2           
L1:move R4, R3
L2:          
```

What if-then-else has many instructions? What if unbalanced?
Full Predication

– Almost all instructions can be executed conditionally under predicate
– Instruction becomes NOP if predicate register false
Full Predication

– Almost all instructions can be executed conditionally under predicate

– Instruction becomes NOP if predicate register false

If

Inst 1
Inst 2
br a==b, b2

Else

Inst 3
Inst 4
br b3

Then

Inst 5
Inst 6

Inst 7
Inst 8

Four basic blocks
Full Predication

– Almost all instructions can be executed conditionally under predicate
– Instruction becomes NOP if predicate register false

Mahlke et al, ISCA95: On average >50% branches removed
movz rd, rs, rt if ( R[rt] == 0 ) then R[rd] <- R[rs]
Predicates and the Datapath

movz rd, rs, rt if ( R[rt] == 0 ) then R[rd] <- R[rs]
• Suppress Writeback
Predicates and the Datapath

movz rd, rs, rt if ( R[rt] == 0 ) then R[rd] <- R[rs]

• Suppress Writeback
• Bypassing value doesn’t work!
  • Need initial value (Extra read port on RF)
Problems with Full Predication

• Adds another register file for predicates
• Need to bypass predicates
• Need original value to use data value bypassing (extra read port on RF)
Leveraging Speculative Execution and Reacting to Dynamic Events in a VLIW

Speculation:
• Moving instructions across branches
• Moving memory operations past other memory operations

Dynamic Events:
• Cache Miss
• Exceptions
• Branch Mispredict
Before Code Motion

MUL R1, R2, R3
ADDIU R11, R10, 1
MUL R5, R1, R4
MUL R7, R5, R6
SW R7, 0(R16)
ADDIU R12, R11, 1
LW R14, 0(R9)
ADD R13, R12, R14
ADD R14, R12, R13
BNEQ R16, target

Code Motion
# Code Motion

<table>
<thead>
<tr>
<th>Before Code Motion</th>
<th>After Code Motion</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL R1, R2, R3</td>
<td>LW R14, 0(R9)</td>
</tr>
<tr>
<td>ADDIU R11,R10,1</td>
<td>MUL R1, R2, R3</td>
</tr>
<tr>
<td>MUL R5, R1, R4</td>
<td>ADDIU R11,R10,1</td>
</tr>
<tr>
<td>MUL R7, R5, R6</td>
<td>ADDIU R12,R11,1</td>
</tr>
<tr>
<td>SW R7, 0(R16)</td>
<td>MUL R5, R1, R4</td>
</tr>
<tr>
<td>ADDIU R12,R11,1</td>
<td>ADD R13,R12,R14</td>
</tr>
<tr>
<td>LW R14, 0(R9)</td>
<td>MUL R7, R5, R6</td>
</tr>
<tr>
<td>ADD R13,R12,R14</td>
<td>ADD R14,R12,R13</td>
</tr>
<tr>
<td>ADD R14,R12,R13</td>
<td>SW R7, 0(R16)</td>
</tr>
<tr>
<td>BNEQ R16, target</td>
<td>BNEQ R16, target</td>
</tr>
</tbody>
</table>
Scheduling and Bundling

Before Bundling

LW R14, 0(R9)
ADDIU R11,R10,1
MUL R1, R2, R3
ADDIU R12,R11,1
MUL R5, R1, R4
ADD R13,R12,R14
MUL R7, R5, R6
ADD R14,R12,R13
SW R7, 0(R16)
BNEQ R16, target

After Bundling

{LW R14, 0(R9)
ADDIU R11,R10,1
MUL R1, R2, R3}
{ADDIU R12,R11,1
  MUL R5, R1, R4}
{ADD R13,R12,R14
  MUL R7, R5, R6}
{ADD R14,R12,R13
  SW R7, 0(R16)
  BNEQ R16, target}
VLIW Speculative Execution

Problem: Branches restrict compiler code motion

Inst 1
Inst 2
br a==b, b2

Load r1
Use r1
Inst 3
VLIW Speculative Execution

Problem: Branches restrict compiler code motion

Can’t move load above branch because might cause spurious exception
VLIW Speculative Execution

Problem: Branches restrict compiler code motion
Solution: Speculative operations that don’t cause exceptions

Inst 1
Inst 2
br a==b, b2

Load r1
Use r1
Inst 3

Can’t move load above branch because might cause spurious exception
VLIW Speculative Execution

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don’t cause exceptions

Can’t move load above branch because might cause spurious exception

Speculative load never causes exception, but sets “poison” bit on destination register

Check for exception in original home block jumps to fixup code if exception detected
VLIW Speculative Execution

Problem: Branches restrict compiler code motion
Solution: Speculative operations that don’t cause exceptions

```
Inst 1
Inst 2
br a==b, b2

Load r1
Use r1
Inst 3

Can’t move load above branch because might cause spurious exception
```

```
Load.s r1
Inst 1
Inst 2
br a==b, b2

Chk.s r1
Use r1
Inst 3

Speculative load never causes exception, but sets “poison” bit on destination register
Check for exception in original home block jumps to fixup code if exception detected
```

Particularly useful for scheduling long latency loads early
VLIW Data Speculation

Problem: Possible memory hazards limit code scheduling
VLIW Data Speculation

Problem: Possible memory hazards limit code scheduling
Solution: Hardware to check pointer hazards
VLIW Data Speculation

Problem: Possible memory hazards limit code scheduling
Solution: Hardware to check pointer hazards

Can’t move load above store because store might be to same address
VLIW Data Speculation

Problem: Possible memory hazards limit code scheduling
Solution: Hardware to check pointer hazards

Can’t move load above store because store might be to same address

```
Inst 1
Inst 2
Store
Load r1
Use r1
Inst 3
```

```
Load.a r1
Inst 1
Inst 2
Store
Load.c
Use r1
Inst 3
```

Data speculative load adds address to address check table

Store invalidates any matching loads in address check table (ALAT)

Check if load invalid (or missing), jump to fix-up code if so
VLIW Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Hardware to check pointer hazards

Can’t move load above store because store might be to same address

Requires associative hardware in address check table
ALAT (Advanced Load Address Table)

<table>
<thead>
<tr>
<th>Store CAM on Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Number</td>
</tr>
<tr>
<td>--------------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

- Load.a adds entry to ALAT
- Store removes entry if address/size match
- Check instruction (chk.a or ld.c) checks to make sure that address is still in ALAT and intermediary store did not push it out.
  - If not found, run recovery code (ex. re-execute load)
VLIW Multi-Way Branches

Problem: Long instructions provide few opportunities for branches
VLIW Multi-Way Branches

Problem: Long instructions provide few opportunities for branches
Solution: Allow one instruction to branch multiple directions
VLIW Multi-Way Branches

Problem: Long instructions provide few opportunities for branches
Solution: Allow one instruction to branch multiple directions

{
  .mii
  cmp.eq P1,P2 = R1, R2
  cmp.ne P3,P4 = R4, R5
  cmp.lt P5,P6, R8, R9
}
{
  .bbb
  (P1) br.cond label1
  (P2) br.cond label2
  (P5) br.cond label3
}
// fall through code here
Scheduling Around Dynamic Events

• Cache Miss
  – Informing loads (loads nullify subsequent instructions)
  – Elbrus (Soviet/Russian) processor had branch on cache miss

• Branch Mispredict
  – Delay slots with predicated instructions

• Exceptions
  – Hard on superscalar also…
Clustered VLIW

- Divide machine into clusters (local register files and local functional units)
- Lower bandwidth between clusters/Higher latency between clusters
- Used in:
  - HP/ST Lx Processor (printers)
  - TI C6x Series DSP

[Image Credit: Texas Instruments Incorporated]
Intel Itanium, EPIC IA-64
Intel Itanium, EPIC IA-64

• EPIC is the style of architecture (CISC, RISC)
  – Explicitly Parallel Instruction Computing
Intel Itanium, EPIC IA-64

• EPIC is the style of architecture (CISC, RISC)
  – Explicitly Parallel Instruction Computing
• IA-64 was Intel’s chosen ISA (x86, MIPS)
  – IA-64 = Intel Architecture 64-bit
  – An object-code-compatible VLIW
Intel Itanium, EPIC IA-64

• EPIC is the style of architecture (CISC, RISC)
  – Explicitly Parallel Instruction Computing
• IA-64 was Intel’s chosen ISA (x86, MIPS)
  – IA-64 = Intel Architecture 64-bit
  – An object-code-compatible VLIW
• Merced was first Itanium implementation (8086)
  – First customer shipment expected 1997 (actually 2001)
  – McKinley, second implementation shipped in 2002
  – Recent version, Poulson, eight cores, 32nm, announced 2011
Eight Core Itanium “Poulson” [Intel 2011]

- 8 cores
- 1-cycle 16KB L1 I&D caches
- 9-cycle 512KB L2 I-cache
- 8-cycle 256KB L2 D-cache
- 32 MB shared L3 cache
- 544mm² in 32nm CMOS
- Over 3 billion transistors
- Cores are 2-way multithreaded
- 6 instruction/cycle fetch
  - Two 128-bit bundles
- Up to 12 insts/cycle execute
- 1.7GHz - 2.53GHz
IA-64 Instruction Format

128-bit instruction bundle

- Template bits describe grouping of these instructions with others in adjacent bundles
- Each group contains instructions that can execute in parallel
IA-64 Registers

- 128 General Purpose 64-bit Integer Registers
- 128 General Purpose 64/80-bit Floating Point Registers
- 64 1-bit Predicate Registers

- GPRs “rotate” to reduce code size for software pipelined loops
  - Rotation is a simple form of register renaming allowing one instruction to address different physical registers on each iteration
Problem: Scheduling of loops requires many register names and duplicated code in prolog and epilog
Solution: Allocate a new set of registers for each loop iteration

Rotating Register Base (RRB) register points to base of current register set. Value added on to logical register specifier to give physical register number. Usually, split into rotating and non-rotating registers.
Rotating Register File

(Previous Loop Example)

<table>
<thead>
<tr>
<th>lw f1, ()</th>
<th>add.s f5, f4, ...</th>
<th>sw f9, ()</th>
<th>bloop</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw P9, ()</td>
<td>add.s P13, P12,</td>
<td>sw P17, ()</td>
<td>bloop</td>
</tr>
<tr>
<td>lw P8, ()</td>
<td>add.s P12, P11,</td>
<td>sw P16, ()</td>
<td>bloop</td>
</tr>
<tr>
<td>lw P7, ()</td>
<td>add.s P11, P10,</td>
<td>sw P15, ()</td>
<td>bloop</td>
</tr>
<tr>
<td>lw P6, ()</td>
<td>add.s P10, P9,</td>
<td>sw P14, ()</td>
<td>bloop</td>
</tr>
<tr>
<td>lw P5, ()</td>
<td>add.s P9, P8,</td>
<td>sw P13, ()</td>
<td>bloop</td>
</tr>
<tr>
<td>lw P4, ()</td>
<td>add.s P8, P7,</td>
<td>sw P12, ()</td>
<td>bloop</td>
</tr>
<tr>
<td>lw P3, ()</td>
<td>add.s P7, P6,</td>
<td>sw P11, ()</td>
<td>bloop</td>
</tr>
<tr>
<td>lw P2, ()</td>
<td>add.s P6, P5,</td>
<td>sw P10, ()</td>
<td>bloop</td>
</tr>
</tbody>
</table>

RRB=8  
RRB=7  
RRB=6  
RRB=5  
RRB=4  
RRB=3  
RRB=2  
RRB=1
Rotating Register File
(Previous Loop Example)

Three cycle load latency encoded as difference of 3 in register specifier number (f4 - f1 = 3)

Four cycle add.s latency encoded as difference of 4 in register specifier number (f9 – f5 = 4)

<table>
<thead>
<tr>
<th>lw f1, ()</th>
<th>add.s f5, f4, ...</th>
<th>sw f9, ()</th>
<th>bloop</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw P9, ()</td>
<td>add.s P13, P12,</td>
<td>sw P17, ()</td>
<td>bloop</td>
</tr>
<tr>
<td>lw P8, ()</td>
<td>add.s P12, P11,</td>
<td>sw P16, ()</td>
<td>bloop</td>
</tr>
<tr>
<td>lw P7, ()</td>
<td>add.s P11, P10,</td>
<td>sw P15, ()</td>
<td>bloop</td>
</tr>
<tr>
<td>lw P6, ()</td>
<td>add.s P10, P9,</td>
<td>sw P14, ()</td>
<td>bloop</td>
</tr>
<tr>
<td>lw P5, ()</td>
<td>add.s P9, P8,</td>
<td>sw P13, ()</td>
<td>bloop</td>
</tr>
<tr>
<td>lw P4, ()</td>
<td>add.s P8, P7,</td>
<td>sw P12, ()</td>
<td>bloop</td>
</tr>
<tr>
<td>lw P3, ()</td>
<td>add.s P7, P6,</td>
<td>sw P11, ()</td>
<td>bloop</td>
</tr>
<tr>
<td>lw P2, ()</td>
<td>add.s P6, P5,</td>
<td>sw P10, ()</td>
<td>bloop</td>
</tr>
</tbody>
</table>

RRB=8
RRB=7
RRB=6
RRB=5
RRB=4
RRB=3
RRB=2
RRB=1
## Rotating Register File

(Previous Loop Example)

<table>
<thead>
<tr>
<th>Instruction 1</th>
<th>Instruction 2</th>
<th>Instruction 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw f1, ()</td>
<td>add.s f5, f4, ...</td>
<td>sw f9, ()</td>
</tr>
<tr>
<td>lw P9, ()</td>
<td>add.s P13, P12,</td>
<td>sw P17, ()</td>
</tr>
<tr>
<td>lw P8, ()</td>
<td>add.s P12, P11,</td>
<td>sw P16, ()</td>
</tr>
<tr>
<td>lw P7, ()</td>
<td>add.s P11, P10,</td>
<td>sw P15, ()</td>
</tr>
<tr>
<td>lw P6, ()</td>
<td>add.s P10, P9,</td>
<td>sw P14, ()</td>
</tr>
<tr>
<td>lw P5, ()</td>
<td>add.s P9, P8,</td>
<td>sw P13, ()</td>
</tr>
<tr>
<td>lw P4, ()</td>
<td>add.s P8, P7,</td>
<td>sw P12, ()</td>
</tr>
<tr>
<td>lw P3, ()</td>
<td>add.s P7, P6,</td>
<td>sw P11, ()</td>
</tr>
<tr>
<td>lw P2, ()</td>
<td>add.s P6, P5,</td>
<td>sw P10, ()</td>
</tr>
</tbody>
</table>

Three cycle load latency encoded as difference of 3 in register specifier number (f4 - f1 = 3)

Four cycle add.s latency encoded as difference of 4 in register specifier number (f9 - f5 = 4)
### Rotating Register File

(Previous Loop Example)

Three cycle load latency encoded as difference of 3 in register specifier number \((f4 - f1 = 3)\)

Four cycle add.s latency encoded as difference of 4 in register specifier number \((f9 – f5 = 4)\)

<table>
<thead>
<tr>
<th>lw f1, ()</th>
<th>add.s f5, f4, ...</th>
<th>sw f9, ()</th>
<th>bloop</th>
</tr>
</thead>
</table>

| lw P9, ()  | add.s P13, P12, | sw P17, () | bloop |
| lw P8, ()  | add.s P12, P11, | sw P16, () | bloop |
| lw P7, ()  | add.s P11, P10, | sw P15, () | bloop |
| lw P6, ()  | add.s P10, P9,  | sw P14, () | bloop |
| lw P5, ()  | add.s P9, P8,   | sw P13, () | bloop |
| lw P4, ()  | add.s P8, P7,   | sw P12, () | bloop |
| lw P3, ()  | add.s P7, P6,   | sw P11, () | bloop |
| lw P2, ()  | add.s P6, P5,   | sw P10, () | bloop |

RRB = 8
RRB = 7
RRB = 6
RRB = 5
RRB = 4
RRB = 3
RRB = 2
RRB = 1
Why Itanium (IA-64) Failed  
(My Opinion Only)

• Many **Architectural (ISA)** visible widgets tied hands of microarchitect designers  
  – ALAT  
  – Predication (increases inter-dependency of instructions)  
  – Rotating Register file  
• First implementations had very low clock rate  
• Complex encoding  
• Code size bloat  
• Did not fundamentally solve some of the dynamic scheduling  
• Large compiler complexity  
  – Profiling needed for really good performance  
• Limited Static Instruction Level Parallelism (ILP)  
• People **did** build more complex superscalars. Area was not the most critical constraint. (Code compatibility and Power were critical)  
• AMD64!
Acknowledgements

• These slides contain material developed and copyright by:
  – Arvind (MIT)
  – Krste Asanovic (MIT/UCB)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
  – David Patterson (UCB)
  – Christopher Batten (Cornell)

• MIT material derived from course 6.823
• UCB material derived from course CS252 & CS152
• Cornell material derived from course ECE 4750