Superscalar Control Logic Scaling

- Each issued instruction must somehow check against $W \times L$ 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, WQ, 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 $L$

$\Rightarrow$ Out-of-order control logic grows faster than $W^2$ ($\sim W^3$)

Out-of-Order Control Complexity:
MIPS R10000

[ SGI/MIPS Technologies Inc., 1995 ]
Sequential ISA Bottleneck

Sequential source code

Superscalar compiler

Find independent operations

Schedule operations

Superscalar processor

Check instruction dependencies

Schedule execution

Sequential machine code

VLIW: Very Long Instruction Word

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two Integer Units, Single Cycle Latency

Two Load/Store Units, Three Cycle Latency

Two Floating-Point Units, Four Cycle Latency

- 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 Processors**

In-order issue of 4 instructions per cycle.

Loop:
- **LW** $r1, 0(r2)$
- **MUL** $r3, r1, r4$
- **SW** $r3, 0(r5)$
- **ADDW** $r2, r2, r4$
- **ADDW** $r5, r5, r1$
- **ADDW** $r7, r7, -1$
- **BHT2** $r7, 100$

**Architecture Exposed**
- Number of functional units
- Functional unit latency
- Branch resolution latency

Compiler must schedule VLIW "instructions" given these constraints.

7 cycles/ITA, 64 ITA to process 64 ELM

448 total cycles.
VLW PIPELINE DIAGRAM

\[ \begin{align*}
0 & \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13 \\
\text{ADDW} & \rightarrow r7, r7 \rightarrow r1, -1 \\
\text{SUB} & \rightarrow r7, 10, -p \\
\text{ADDW} & \rightarrow r5, r5, -1 \\
\text{SW} & \rightarrow r3, 0, 0(r3) \\
\text{MUL} & \rightarrow r3, r1, r7 \\
\text{ADDW} & \rightarrow r3, r2, r2, r1 \\
\text{SUB} & \rightarrow r1, 0(r2) \\
\text{ADDW} & \rightarrow r2, r3, r2, r1 \\
\end{align*} \]
Loop Unrolling:

Unroll loop to amortize loop overhead, reduce # of iterations.

Loop:
- \( w \), \( r1, 0(r2) \)
- \( w \), \( r3, 4(r2) \)
- \( w \), \( r5, 8(r2) \)
- \( w \), \( r6, 12(r2) \)
- \( \text{mul} \), \( r8, r1, r4 \)
- \( \text{mul} \), \( r9, r3, r4 \)
- \( \text{mul} \), \( r10, r5, r4 \)
- \( \text{mul} \), \( r11, r6, r4 \)
- \( \text{sw} \), \( r8, 0(r5) \)
- \( \text{sw} \), \( r9, 4(r5) \)
- \( \text{sw} \), \( r10, 8(r5) \)
- \( \text{sw} \), \( r11, 12(r5) \)
- \( \text{addu} \), \( r2, r2, 16 \)
- \( \text{addu} \), \( r3, r3, 16 \)
- \( \text{addu} \), \( r7, r7, 4 \)
- \( \text{bjt2} \), \( r7, \text{loop} \)

Y-Pipe: |
---|---|---|---|
\text{mul} \, r8 \n\text{mul} \, r9 \n\text{addu} \, r7, -4 \n\text{bjt2} \, r7, \text{loop} \n
X-Pipe: |
---|---|---|---|
\text{addu} \, r7, -4 \n\text{bjt2} \, r7 \n\text{addu} \, r2 \n\text{addu} \, r5 \n
L-Pipe: |
---|---|---|---|
\text{sw} \, r8 \n\text{sw} \, r9 \n\text{sw} \, r10 \n\text{sw} \, r11 \n
S-Pipe: |
---|---|---|---|
\text{sw} \, r8 \n\text{sw} \, r9 \n\text{sw} \, r10 \n\text{sw} \, r11 \n
10 cycles/iteration, 16 iterations to process 64 elements
160 total cycles

Unroll by factor of eight?

Y-Pipe: |
---|---|---|---|
\text{mul} \n\text{mul} \n\text{addu} \, r7, -4 \n\text{bjt2} \, r7, \text{loop} \n
X-Pipe: |
---|---|---|---|
\text{mul} \n\text{mul} \n\text{addu} \, r2 \n\text{addu} \, r5 \n
L-Pipe: |
---|---|---|---|
\text{sw} \n\text{sw} \n\text{sw} \n\text{sw} \n
S-Pipe: |
---|---|---|---|
\text{sw} \n\text{sw} \n\text{sw} \n\text{sw} \n
14 cycles/iteration, 8 iterations to process 64 elements
112 total cycles

Finally got to peak: 1 CD, 1 ST, 1 MUL per cycle.
SOFTWARE PIPELINING

TAKE INSTRUCTIONS FROM MULTIPLE ITERATIONS TO CREATE NEW
INSTRUCTION STREAMS THAT CAN RUN AT HIGHER PEAK THROUGHPUT

prologue:

\[
\begin{align*}
\text{lw} & \ r1, 0(r2) \\
\text{mul} & \ r3, r1, r4 \\
\text{sw} & \ r1, 0(r3) \\
\end{align*}
\]

loop:

\[
\begin{align*}
\text{sw} & \ r3, 0(r5) \\
\text{mul} & \ r3, r1, r4 \\
\text{lw} & \ r1, 0(r3) \\
\end{align*}
\]

\[
\begin{align*}
\text{lw} & \ r1, 0(r2) \\
\text{mul} & \ r3, r1, r4 \\
\text{addw} & \ r2, r2, r4 \\
\text{addw} & \ r3, r5, r4 \\
\text{addw} & \ r3, r3, -1 \\
\text{bne} & \ r3, \ loop \\
\end{align*}
\]

epilogue:

\[
\begin{align*}
\text{sw} & \ r3, 0(r5) \\
\text{addw} & \ r5, r5, r4 \\
\text{mul} & \ r3, r1, r4 \\
\text{sw} & \ r3, 0(r3) \\
\end{align*}
\]

ORIGINAL

loop:

\[
\begin{align*}
\text{lw} & \ r1, 0(r2) \\
\text{mul} & \ r3, r1, r4 \\
\text{sw} & \ r3, 0(r5) \\
\text{addw} & \ r2, r2, r4 \\
\text{addw} & \ r3, r5, r4 \\
\text{addw} & \ r3, r3, -1 \\
\text{bne} & \ r3, \ loop \\
\end{align*}
\]

pipeline startup

itr 0

\[
\text{lw} \times 3w
\]

itr 1

\[
\text{lw} \times 5w
\]

itr 2

\[
\text{lw} \times 3w
\]

itr 3

\[
\text{lw} \times 5w
\]

Achieve full throughput

Start second itr before first iteration is finished

Let pipeline drain
Sw pipelining produces more compact code, uses less registers, and can deal with irregularly sized input arrays better than loop unrolling.

Sw pipelining does not require loop overhead (essentially same # of iterations)

Sw pipelining allows code to quickly get up to peak through prologue and epilogue at pre loop.

```
<table>
<thead>
<tr>
<th>Y-PIPE</th>
<th>X-PIPE</th>
<th>L-PIPE</th>
<th>S-PIPE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL r3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ADD r2</td>
<td>LW r1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>LW r1</td>
<td></td>
</tr>
<tr>
<td>MUL r3</td>
<td>ADD r3</td>
<td>LW r1</td>
<td>SW r3</td>
</tr>
<tr>
<td></td>
<td>LW r2</td>
<td>ADD r2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL r3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD r5</td>
<td></td>
<td></td>
<td>SW r3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Prologue = 6 cycles
Main loop = 4 cycles /1 loop, 62 loops = 248 cycles
Epilogue = 5 cycles

Total = 259 cycles
loop unrolling + SW pipelining

Use loop unrolling to minimize loop overhead.
Use SW pipelining to reduce reg pressure, code size, + flexibility

First unroll loop

loop:

\[
\begin{align*}
&\text{lw} r1, 0(r2) \\
&\text{lw} r2, 4(r2) \\
&\text{lw} r3, 8(r2) \\
&\text{lw} r6, 12(r2) \\
&\text{mul} r8, r1, r4 \\
&\text{mul} r9, r3, r4 \\
&\text{mul} r10, r5, r4 \\
&\text{mul} r11, r6, r4 \\
&\text{sw} r8, 0(r8) \\
&\text{sw} r9, 4(r8) \\
&\text{sw} r10, 8(r8) \\
&\text{sw} r11, 12(r8) \\
&\text{addw} r2, r2, 16 \\
&\text{addw} r5, r5, 16 \\
&\text{addw} r7, r7, -4 \\
&\text{b} r7, \text{loop}
\end{align*}
\]
<table>
<thead>
<tr>
<th>Y-PIPE</th>
<th>X-PIPE</th>
<th>L-PIPE</th>
<th>S-PIPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>mul r8</td>
<td>addw r2</td>
<td>lw r1</td>
<td>lw r1</td>
</tr>
<tr>
<td>mul r9</td>
<td></td>
<td>lw r3</td>
<td>lw r3</td>
</tr>
<tr>
<td>mul r10</td>
<td></td>
<td>lw r5</td>
<td>lw r5</td>
</tr>
<tr>
<td>mul r11</td>
<td></td>
<td>lw r6</td>
<td>lw r6</td>
</tr>
</tbody>
</table>

Main Loop:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mul r8</td>
<td>addw rt</td>
<td>lw r1, 0(r1)</td>
<td>sw r8, 0(r8)</td>
</tr>
<tr>
<td>mul r9</td>
<td>addw rt</td>
<td>lw r3, 4(r3)</td>
<td>sw r9, 4(r9)</td>
</tr>
<tr>
<td>mul r10</td>
<td>addw r16</td>
<td>lw r5, 8(r5)</td>
<td>sw r10, 8(r10)</td>
</tr>
<tr>
<td>mul r11</td>
<td>addw r16</td>
<td>lw r6, 12(r6)</td>
<td>sw r11, 12(r11)</td>
</tr>
</tbody>
</table>

Epilogue:

<table>
<thead>
<tr>
<th></th>
<th>lw r5</th>
<th>sw r5, 0(r5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lw r6</td>
<td>sw r6, 4(r6)</td>
</tr>
<tr>
<td></td>
<td>lw r7</td>
<td>sw r7, 0(r7)</td>
</tr>
<tr>
<td></td>
<td>lw r8</td>
<td>sw r8, 4(r8)</td>
</tr>
<tr>
<td></td>
<td>lw r9</td>
<td>sw r9, 0(r9)</td>
</tr>
<tr>
<td></td>
<td>lw r10</td>
<td>sw r10, 4(r10)</td>
</tr>
<tr>
<td></td>
<td>lw r11</td>
<td>sw r11, 0(r11)</td>
</tr>
</tbody>
</table>

Notice clever use of offset.

Might still need fix up code in case input not multiple of 4.

Prologue = 9 cycles
Main Loop = 4 cycles / 16x = 64 cycles
Epilogue = 8 cycles
Total = 73 cycles!
Software Pipelining & Loop Unrolling vs. Loop Unrolling Alone

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
- Only one branch per VLIW instruction
- Difficult to find ILP in individual basic blocks
Trace Scheduling

- 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

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
**VLIW Instruction Encoding**

- Schemes to reduce effect of unused fields
  - Compressed format in memory, expand on I-cache refill
  - Mark parallel groups
  - Provide a single-op VLIW instruction

---

**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
  beq   r1, r0, L1
else
  move r4, r2
  j      L2
L1:
  move r4, r3
L2:
```

What if then else has many instructions? Or is unbalanced?
**Full Predication**

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

| b0: | Inst 1     |   if   | Inst 1 |
|     | Inst 2     |         | Inst 2 |
|     | br a==b, b2 |         |       |

| b1: | Inst 3     | else | Inst 4     |
|     | Inst 4     |       | br b3       |

| b2: | Inst 5     | then | Inst 6     |
|     | Inst 6     |       |             |

| b3: | Inst 7     |       | Inst 8     |
|     | Inst 8     |       |             |

Four basic blocks

**One basic block**

<table>
<thead>
<tr>
<th>Inst 1</th>
<th>Inst 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1,p2 &lt;- cmp(a==b)</td>
<td></td>
</tr>
<tr>
<td>(p1) Inst 3</td>
<td></td>
</tr>
<tr>
<td>(p1) Inst 4</td>
<td></td>
</tr>
<tr>
<td>Inst 7</td>
<td></td>
</tr>
</tbody>
</table>

*Mahlke et al, ISCA95: On average >50% branches removed*

---

**Rotating Register File**

Problems: Scheduled loops require lots of registers,
Lots of duplicated code in prolog, epilog

Solution: Allocate new set of registers for each loop iteration

| RR=3 |
|      |
|      |
| P0   |
| P1   |
| P2   |
| P3   |
| P4   |
| P5   |
| P6   |
| P7   |

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.
Memory Latency Register (MLR)

Problem: Loads have variable latency
Solution: Let software choose desired memory latency

- Compiler schedules code for maximum load-use distance
- Software sets MLR to latency that matches code schedule
- Hardware ensures that loads take exactly MLR cycles to return values into processor pipeline
  - Hardware buffers loads that return early
  - Hardware stalls processor if loads return late