Microcoded and VLIW Processors

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.
Hardwired vs Microcoded Processors

- All processors we have seen so far are hardwired: The microarchitecture directly implements all the instructions in the ISA.
Hardwired vs Microcoded Processors

- All processors we have seen so far are hardwired: The microarchitecture directly implements all the instructions in the ISA.

- Microcoded processors add a layer of interpretation: Each ISA instruction is executed as a sequence of simpler *microinstructions*:
  - Simpler implementation
  - Lower performance than hardwired (*CPI > 1*)
Hardwired vs Microcoded Processors

- All processors we have seen so far are hardwired: The microarchitecture directly implements all the instructions in the ISA

- Microcoded processors add a layer of interpretation: Each ISA instruction is executed as a sequence of simpler microinstructions
  - Simpler implementation
  - Lower performance than hardwired (CPI > 1)

- Microcoding common until the 80s, still in use today (e.g., complex x86 instructions are decoded into multiple “micro-ops”)
Microcontrol Unit
[Maurice Wilkes, 1954]

Embed the control logic state table in a read-only memory array

op code
conditional flip-flop
μ address

Decoder

Matrix A
Matrix B

Control lines to ALU, MUXs, Registers

Next state
Microcoded Microarchitecture

μcontroller (ROM)

Datapath

Memory (RAM)

Data

Addr

enMem
MemWrt

busy?
zero?
opcode

April 23, 2020

MIT 6.823 Spring 2020
Microcoded Microarchitecture

Memory (RAM)

Datapath

μcontroller (ROM)

holds fixed microcode instructions

busy?

zero?

opcode

Data

Addr

enMem

MemWrt

April 23, 2020

MIT 6.823 Spring 2020
Microcoded Microarchitecture

- **μcontroller (ROM)**
  - busy?
  - zero?
  - opcode
  - holds fixed microcode instructions

- **Datapath**
  - Data
  - Addr

- **Memory (RAM)**
  - enMem
  - MemWrt
  - holds user program written in macrocode instructions (e.g., MIPS, x86, etc.)

- **Rambler**

- **EnMem**

- **enMem**

- **MemWrt**

- **Busy?**

- **Zero?**

- **Opcode**
A Bus-based Datapath for MIPS

Diagram: Flowchart of bus-based datapath with components labeled as follows:
- OpSel
- IdA
- IdB
- ALU control
- enALU
- ALU
- zero?
- Bus 32
A Bus-based Datapath for MIPS
A Bus-based Datapath for MIPS
A Bus-based Datapath for MIPS

[Diagram of a bus-based datapath for MIPS]

- Opcode
  - ldIR
- OpSel
- IdA
- IdB
- zero?
- 32(PC)
- 31(Link)
- rd
- rt
- rs
- RegSel
- addr
- 32 GPRs + PC ...
- 32-bit Reg
- data
- Addr
- Memory
- busy
- ldMA
- MA
- Addr
- data
- MemWrt
- enMem
- ExtSel
- Imm
- Ext
- enImm
- ALU
- control
- enALU
- rd
- rt
- rs
- Bus
- 32
A Bus-based Datapath for MIPS

Microinstruction: register to register transfer (17 control signals)

MA \leftarrow PC \quad \text{means} \quad \text{RegSel} = \text{PC}; \quad \text{enReg}=\text{yes}; \quad \text{IdMA}=\text{yes}

B \leftarrow \text{Reg}[rt] \quad \text{means}
A Bus-based Datapath for MIPS

Microinstruction: register to register transfer (17 control signals)

MA ← PC means RegSel = PC; enReg = yes; IdMA = yes
B ← Reg[rt] means RegSel = rt; enReg = yes; IdB = yes

MA  PC
RegSel = PC; enReg = yes; IdMA = yes

B  Reg[rt]
RegSel = rt; enReg = yes; IdB = yes
Memory Module

- Assumption: Memory operates asynchronously and is slow compared to Reg-to-Reg transfers
Microcode Controller

- Opcode → ext
- absolute
- op-group
- \( \muPC \) \( \muPC+1 \)
- \( \muPC (\text{state}) \)
- address
- data
- Control ROM
- Control Signals (17)
- input encoding reduces ROM height
- jump logic
- zero
- busy
- input encoding reduces ROM height
Microcode Controller

Control Signals (17)

 Opcode → ext

op-group

absolute

μPC

μPC+1

+1

μPC Src

jump logic

Control ROM

address

data

Control Signals (17)

input encoding reduces ROM height

next-state encoding reduces ROM width

zero

busy

MIT 6.823 Spring 2020
Microcode Controller

\[ \mu \text{JumpType} = \text{next} | \text{spin} | \text{fetch} | \text{dispatch} | \text{feqz} | \text{fnez} \]

- Opcode → ext
- absolute
- \( \mu \text{PC} \) → \( \mu \text{PC+1} \)
- \( \mu \text{PC} \) (state)
- address
- data

Control Signals (17)

next-state encoding reduces ROM width

input encoding reduces ROM height

Opcode

ext

op-group

jump logic

+1

\( \mu \text{PCSrc} \)

zero

busy

jump

logic

\( \mu \text{PC} \)

\( \mu \text{PC} + 1 \)
<table>
<thead>
<tr>
<th>μPCSrc</th>
<th>Case</th>
<th>μJumpTypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>⇒</td>
<td>μPC+1</td>
</tr>
<tr>
<td>spin</td>
<td>⇒</td>
<td>if (busy) then μPC else μPC+1</td>
</tr>
<tr>
<td>fetch</td>
<td>⇒</td>
<td>absolute</td>
</tr>
<tr>
<td>dispatch</td>
<td>⇒</td>
<td>op-group</td>
</tr>
<tr>
<td>fpeqz</td>
<td>⇒</td>
<td>if (zero) then absolute else μPC+1</td>
</tr>
<tr>
<td>fnez</td>
<td>⇒</td>
<td>if (zero) then μPC+1 else absolute</td>
</tr>
</tbody>
</table>
Instruction Execution

Execution of a MIPS instruction involves

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. memory operation (optional)
5. write back to register file (optional)  
   + the computation of the
   next instruction address
# Instruction Fetch

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch(_0)</td>
<td>MA $\leftarrow$ PC</td>
<td></td>
</tr>
<tr>
<td>fetch(_1)</td>
<td>IR $\leftarrow$ Memory</td>
<td></td>
</tr>
<tr>
<td>fetch(_2)</td>
<td>A $\leftarrow$ PC</td>
<td></td>
</tr>
<tr>
<td>fetch(_3)</td>
<td>PC $\leftarrow$ A + 4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU(_0)</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td></td>
</tr>
<tr>
<td>ALU(_1)</td>
<td>B $\leftarrow$ Reg[rt]</td>
<td></td>
</tr>
<tr>
<td>ALU(_2)</td>
<td>Reg[rd] $\leftarrow$ func(A,B)</td>
<td></td>
</tr>
<tr>
<td>ALU(_i)_0</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td></td>
</tr>
<tr>
<td>ALU(_i)_1</td>
<td>B $\leftarrow$ sExt(_{16})(Imm)</td>
<td></td>
</tr>
<tr>
<td>ALU(_i)_2</td>
<td>Reg[rd] $\leftarrow$ Op(A,B)</td>
<td></td>
</tr>
</tbody>
</table>
# Instruction Fetch

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch_0</td>
<td>MA ← PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch_1</td>
<td>IR ← Memory</td>
<td></td>
</tr>
<tr>
<td>fetch_2</td>
<td>A ← PC</td>
<td></td>
</tr>
<tr>
<td>fetch_3</td>
<td>PC ← A + 4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU_0</td>
<td>A ← Reg[rs]</td>
<td></td>
</tr>
<tr>
<td>ALU_1</td>
<td>B ← Reg[rt]</td>
<td></td>
</tr>
<tr>
<td>ALU_2</td>
<td>Reg[rd] ← func(A,B)</td>
<td></td>
</tr>
<tr>
<td>ALU_i_0</td>
<td>A ← Reg[rs]</td>
<td></td>
</tr>
<tr>
<td>ALU_i_1</td>
<td>B ← sExt_16(Imm)</td>
<td></td>
</tr>
<tr>
<td>ALU_i_2</td>
<td>Reg[rd] ← Op(A,B)</td>
<td></td>
</tr>
</tbody>
</table>
# Instruction Fetch

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch(_0)</td>
<td>MA ← PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch(_1)</td>
<td>IR ← Memory</td>
<td>spin</td>
</tr>
<tr>
<td>fetch(_2)</td>
<td>A ← PC</td>
<td></td>
</tr>
<tr>
<td>fetch(_3)</td>
<td>PC ← A + 4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU(_0)</td>
<td>A ← Reg[rs]</td>
<td></td>
</tr>
<tr>
<td>ALU(_1)</td>
<td>B ← Reg[rt]</td>
<td></td>
</tr>
<tr>
<td>ALU(_2)</td>
<td>Reg[rd]←func(A,B)</td>
<td></td>
</tr>
<tr>
<td>ALUi(_0)</td>
<td>A ← Reg[rs]</td>
<td></td>
</tr>
<tr>
<td>ALUi(_1)</td>
<td>B ← sExt(_{16})(Imm)</td>
<td></td>
</tr>
<tr>
<td>ALUi(_2)</td>
<td>Reg[rd]← Op(A,B)</td>
<td></td>
</tr>
</tbody>
</table>
## Instruction Fetch

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch&lt;sub&gt;0&lt;/sub&gt;</td>
<td>MA ← PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch&lt;sub&gt;1&lt;/sub&gt;</td>
<td>IR ← Memory</td>
<td>spin</td>
</tr>
<tr>
<td>fetch&lt;sub&gt;2&lt;/sub&gt;</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch&lt;sub&gt;3&lt;/sub&gt;</td>
<td>PC ← A + 4</td>
<td></td>
</tr>
</tbody>
</table>

... |

| ALU<sub>0</sub> | A ← Reg[rs] | |
| ALU<sub>1</sub> | B ← Reg[rt] | |
| ALU<sub>2</sub> | Reg[rd] ← func(A,B) | |

| ALUi<sub>0</sub> | A ← Reg[rs] | |
| ALUi<sub>1</sub> | B ← sExt<sub>16</sub>(Imm) | |
| ALUi<sub>2</sub> | Reg[rd] ← Op(A,B) | |
# Instruction Fetch

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch₀</td>
<td>MA ← PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch₁</td>
<td>IR ← Memory</td>
<td>spin</td>
</tr>
<tr>
<td>fetch₂</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch₃</td>
<td>PC ← A + 4</td>
<td>dispatch</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU₀</td>
<td>A ← Reg[rs]</td>
<td></td>
</tr>
<tr>
<td>ALU₁</td>
<td>B ← Reg[rt]</td>
<td></td>
</tr>
<tr>
<td>ALU₂</td>
<td>Reg[rd] ← func(A,B)</td>
<td></td>
</tr>
<tr>
<td>ALUᵢ₀</td>
<td>A ← Reg[rs]</td>
<td></td>
</tr>
<tr>
<td>ALUᵢ₁</td>
<td>B ← sExt₁₆(Imm)</td>
<td></td>
</tr>
<tr>
<td>ALUᵢ₂</td>
<td>Reg[rd] ← Op(A,B)</td>
<td></td>
</tr>
</tbody>
</table>
# Instruction Fetch

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch(_0)</td>
<td>MA $\leftarrow$ PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch(_1)</td>
<td>IR $\leftarrow$ Memory</td>
<td>spin</td>
</tr>
<tr>
<td>fetch(_2)</td>
<td>A $\leftarrow$ PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch(_3)</td>
<td>PC $\leftarrow$ A + 4</td>
<td>dispatch</td>
</tr>
</tbody>
</table>

...  

| ALU\(_0\) | A $\leftarrow$ Reg[rs] | next |
| ALU\(_1\) | B $\leftarrow$ Reg[rt] | next |
| ALU\(_2\) | Reg[rd]$\leftarrow$func(A,B) | next |

| ALU\(_i_0\) | A $\leftarrow$ Reg[rs] |
| ALU\(_i_1\) | B $\leftarrow$ sExt\(_{16}\)(Imm) |
| ALU\(_i_2\) | Reg[rd]$\leftarrow$ Op(A,B) |
## Instruction Fetch

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch\textsubscript{0}</td>
<td>MA $\leftarrow$ PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch\textsubscript{1}</td>
<td>IR $\leftarrow$ Memory</td>
<td>spin</td>
</tr>
<tr>
<td>fetch\textsubscript{2}</td>
<td>A $\leftarrow$ PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch\textsubscript{3}</td>
<td>PC $\leftarrow$ A + 4</td>
<td>dispatch</td>
</tr>
<tr>
<td>...</td>
<td>\textellipsis</td>
<td>\textellipsis</td>
</tr>
<tr>
<td>ALU\textsubscript{0}</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALU\textsubscript{1}</td>
<td>B $\leftarrow$ Reg[rt]</td>
<td>next</td>
</tr>
<tr>
<td>ALU\textsubscript{2}</td>
<td>Reg[rd] $\leftarrow$ func(A,B)</td>
<td>\textellipsis</td>
</tr>
<tr>
<td>ALU\textsubscript{i0}</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td>\textellipsis</td>
</tr>
<tr>
<td>ALU\textsubscript{i1}</td>
<td>B $\leftarrow$ sExt\textsubscript{16}(Imm)</td>
<td>\textellipsis</td>
</tr>
<tr>
<td>ALU\textsubscript{i2}</td>
<td>Reg[rd] $\leftarrow$ Op(A,B)</td>
<td>\textellipsis</td>
</tr>
</tbody>
</table>
## Instruction Fetch

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch_0</td>
<td>MA $\leftarrow$ PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch_1</td>
<td>IR $\leftarrow$ Memory</td>
<td>spin</td>
</tr>
<tr>
<td>fetch_2</td>
<td>A $\leftarrow$ PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch_3</td>
<td>PC $\leftarrow$ A + 4</td>
<td>dispatch</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU_0</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALU_1</td>
<td>B $\leftarrow$ Reg[rt]</td>
<td>next</td>
</tr>
<tr>
<td>ALU_2</td>
<td>Reg[rd] $\leftarrow$ func(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>ALUi_0</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td></td>
</tr>
<tr>
<td>ALUi_1</td>
<td>B $\leftarrow$ sExt(_{16})(Imm)</td>
<td></td>
</tr>
<tr>
<td>ALUi_2</td>
<td>Reg[rd] $\leftarrow$ Op(A,B)</td>
<td></td>
</tr>
</tbody>
</table>
# Instruction Fetch

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch₀</td>
<td>MA ← PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch₁</td>
<td>IR ← Memory</td>
<td>spin</td>
</tr>
<tr>
<td>fetch₂</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch₃</td>
<td>PC ← A + 4</td>
<td>dispatch</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALU₁</td>
<td>B ← Reg[rt]</td>
<td>next</td>
</tr>
<tr>
<td>ALU₂</td>
<td>Reg[rd] ← func(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>ALUᵢ₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALUᵢ₁</td>
<td>B ← sExt₁₆(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>ALUᵢ₂</td>
<td>Reg[rd] ← Op(A,B)</td>
<td>fetch</td>
</tr>
</tbody>
</table>
## Load & Store

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>LW₁</td>
<td>B ← sExt₁₆(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>LW₂</td>
<td>MA ← A+B</td>
<td>next</td>
</tr>
<tr>
<td>LW₃</td>
<td>Reg[rt] ← Memory</td>
<td>spin</td>
</tr>
<tr>
<td>LW₄</td>
<td></td>
<td>fetch</td>
</tr>
<tr>
<td>SW₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>SW₁</td>
<td>B ← sExt₁₆(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>SW₂</td>
<td>MA ← A+B</td>
<td>next</td>
</tr>
<tr>
<td>SW₃</td>
<td>Memory ← Reg[rt]</td>
<td>spin</td>
</tr>
<tr>
<td>SW₄</td>
<td></td>
<td>fetch</td>
</tr>
</tbody>
</table>
## Branches

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQZ₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ₁</td>
<td></td>
<td>fnez</td>
</tr>
<tr>
<td>BEQZ₂</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ₃</td>
<td>B ← sExt₁₆(Imm&lt;&lt;2)</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ₄</td>
<td>PC ← A+B</td>
<td>fetch</td>
</tr>
<tr>
<td>BNEZ₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ₁</td>
<td></td>
<td>feqz</td>
</tr>
<tr>
<td>BNEZ₂</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ₃</td>
<td>B ← sExt₁₆(Imm&lt;&lt;2)</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ₄</td>
<td>PC ← A+B</td>
<td>fetch</td>
</tr>
</tbody>
</table>
## Jumps

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>J0</td>
<td>A $\leftarrow$ PC</td>
<td>next</td>
</tr>
<tr>
<td>J1</td>
<td>B $\leftarrow$ IR</td>
<td>next</td>
</tr>
<tr>
<td>J2</td>
<td>PC $\leftarrow$ JumpTarg(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>JR0</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>JR1</td>
<td>PC $\leftarrow$ A</td>
<td>fetch</td>
</tr>
<tr>
<td>JAL0</td>
<td>A $\leftarrow$ PC</td>
<td>next</td>
</tr>
<tr>
<td>JAL1</td>
<td>Reg[31] $\leftarrow$ A</td>
<td>next</td>
</tr>
<tr>
<td>JAL2</td>
<td>B $\leftarrow$ IR</td>
<td>next</td>
</tr>
<tr>
<td>JAL3</td>
<td>PC $\leftarrow$ JumpTarg(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>JALR0</td>
<td>A $\leftarrow$ PC</td>
<td>next</td>
</tr>
<tr>
<td>JALR1</td>
<td>B $\leftarrow$ Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>JALR2</td>
<td>Reg[31] $\leftarrow$ A</td>
<td>next</td>
</tr>
<tr>
<td>JALR3</td>
<td>PC $\leftarrow$ B</td>
<td>fetch</td>
</tr>
</tbody>
</table>
VAX 11-780 Microcode (1978)
Very Long Instruction Word (VLIW) Processors
Sequential ISA Bottleneck

Sequential source code

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

Superscalar compiler

Find independent operations

Schedule operations

Sequential machine code
Sequential ISA Bottleneck

Sequential source code

Superscalar compiler

Find independent operations

Schedule operations

Sequential machine code

Superscalar processor

Check instruction dependencies

April 23, 2020
Sequential ISA Bottleneck

Sequential source code

Superscalar compiler

Sequential machine code

Superscalar processor

Find independent operations

Schedule operations

Check instruction dependencies

Schedule execution

a = foo(b);
for (i=0, i<
VLIW: Very Long Instruction Word

Two Integer Units, Single Cycle Latency

Two Load/Store Units, Three Cycle Latency

Two Floating-Point Units, Four Cycle Latency
VLIW: Very Long Instruction Word

- Multiple operations packed into one instruction
VLIW: Very Long Instruction Word

- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
VLIW: Very Long Instruction Word

- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
VLIW Design Principles

The architecture:
- Allows operation parallelism within an instruction
  - No cross-operation RAW check
- Provides deterministic latency for all operations
  - Latency measured in ‘instructions’
  - No data use allowed before specified latency with no data interlocks

The compiler:
VLIW Design Principles

The architecture:
• Allows operation parallelism within an instruction
  – No cross-operation RAW check
• Provides deterministic latency for all operations
  – Latency measured in ‘instructions’
  – No data use allowed before specified latency with no data interlocks

The compiler:
• Schedules (reorders) to maximize parallel execution
VLIW Design Principles

The architecture:

- Allows operation parallelism within an instruction
  - No cross-operation RAW check
- Provides deterministic latency for all operations
  - Latency measured in ‘instructions’
  - No data use allowed before specified latency with no data interlocks

The compiler:

- Schedules (reorders) to maximize parallel execution
- Guarantees intra-instruction parallelism
VLIW Design Principles

The architecture:
- Allows operation parallelism within an instruction
  - No cross-operation RAW check
- Provides deterministic latency for all operations
  - Latency measured in ‘instructions’
  - No data use allowed before specified latency with no data interlocks

The compiler:
- Schedules (reorders) to maximize parallel execution
- Guarantees intra-instruction parallelism
- Schedules to avoid data hazards (no interlocks)
  - Typically separates operations with explicit NOPs
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
for (i=0; i<N; i++)

Compile

loop:  ld f1, 0(r1)
    add r1, 8
    fadd f2, f0, f1
    sd f2, 0(r2)
    add r2, 8
    bne r1, r3, loop

Schedule

<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:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Loop Execution

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

```
loop:
    ld f1, 0(r1)
    add r1, 8
    fadd f2, f0, f1
    sd f2, 0(r2)
    add r2, 8
    bne r1, r3, loop
```

<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:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Loop Execution

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

Compile

loop: ld f1, 0(r1)
    add r1, 8
    fadd f2, f0, f1
    sd f2, 0(r2)
    add r2, 8
    bne r1, r3, loop

Schedule

<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</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r1</td>
<td>ld</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<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>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
for (i=0; i<N; i++)

loop:  ld f1, 0(r1)
       add r1, 8
       fadd f2, f0, f1
       sd f2, 0(r2)
       add r2, 8
       bne r1, r3, loop
Loop Execution

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

Compile

loop:  ld f1, 0(r1)
   add r1, 8
   fadd f2, f0, f1
   sd f2, 0(r2)
   add r2, 8
   bne r1, r3, loop

<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:</td>
<td>add r1</td>
<td>ld</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r1, 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f2, f0, f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sd f2, 0(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r2, 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne r1, r3, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Schedule

fadd
Loop Execution

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

Compile

loop: ld f1, 0(r1)
    add r1, 8
    fadd f2, f0, f1
    sd f2, 0(r2)
    add r2, 8
    bne r1, r3, loop

Schedule

<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</td>
<td>add r1</td>
<td>ld</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<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>
<td></td>
</tr>
</tbody>
</table>
Loop Execution

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

Compile

loop:  ld f1, 0(r1)
       add r1, 8
       fadd f2, f0, f1
       sd f2, 0(r2)
       add r2, 8
       bne r1, r3, loop

Schedule

<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>add r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd</td>
<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>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
for (i=0; i<N; i++)

Compile
loop:  ld f1, 0(r1)
       add r1, 8
       fadd f2, f0, f1
       sd f2, 0(r2)
       add r2, 8
       bne r1, r3, loop
# Loop Execution

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

### Compile

```c
loop:  ld f1, 0(r1)
    add r1, 8
    fadd f2, f0, f1
    sd f2, 0(r2)
    add r2, 8
    bne r1, r3, 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>add r1</td>
<td>ld</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>fadd</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>add r2</td>
<td>bne</td>
<td>sd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

---

April 23, 2020

MIT 6.823 Spring 2020
Loop Execution

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

Compile

loop: ld f1, 0(r1)
    add r1, 8
    fadd f2, f0, f1
    sd f2, 0(r2)
    add r2, 8
    bne r1, r3, loop

Schedule

How many FP ops/cycle?
Loop Execution

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

Compile

loop:  ld f1, 0(r1)
       add r1, 8
       fadd f2, f0, f1
       sd f2, 0(r2)
       add r2, 8
       bne r1, r3, loop

Schedule

loop:  
       add r1  ld
       fadd
       add r2  bne  sd

How many FP ops/cycle?
1 fadd / 8 cycles = 0.125
Loop Unrolling

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

Unroll inner loop to perform 4 iterations at once

for (i=0; i<N; i+=4)
{
}
Unroll inner loop to perform 4 iterations at once

Is this code correct?
Loop Unrolling

Unroll inner loop to perform 4 iterations at once

Is this code correct?

No, need to handle values of N that are not multiples of unrolling factor with final cleanup loop
Scheduling Loop Unrolled Code

Unroll 4 ways

loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, loop

Schedule
Scheduling Loop Unrolled Code

Unroll 4 ways

Unroll the loop 4 ways:

```
loop:  ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, loop
```

Schedule

Create a schedule table:

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

loop:  ld f1
```

Scheduling Loop Unrolled Code

Unroll 4 ways

loop: ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      sd f8, 24(r2)
      add r2, 32
      bne r1, r3, 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></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>loop:</td>
<td>ld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

April 23, 2020
Scheduling Loop Unrolled Code

Unroll 4 ways

```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, loop
```
Scheduling Loop Unrolled Code

Unroll 4 ways

```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop:</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>
<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>
<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>
Scheduling Loop Unrolled Code

Unroll 4 ways

```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop:</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>
<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>
```
Scheduling Loop Unrolled Code

Unroll 4 ways

loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, loop

Schedule

Int1  Int2  M1  M2  FP+  FPx

loop:  ld f1  ld f2  ld f3  fadd f5  fadd f6  fadd f7  fadd f8
       ld f4  fadd f5  fadd f6  fadd f7  fadd f8
       sd f5  sd f6  sd f7  sd f8
# Scheduling Loop Unrolled Code

## Unroll 4 ways

```plaintext
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, loop
```

## Schedule

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Scheduling Loop Unrolled Code

## Unroll 4 ways

**loop:**
- `ld f1, 0(r1)`
- `ld f2, 8(r1)`
- `ld f3, 16(r1)`
- `ld f4, 24(r1)`
- `add r1, 32`
- `fadd f5, f0, f1`
- `fadd f6, f0, f2`
- `fadd f7, f0, f3`
- `fadd f8, f0, f4`
- `sd f5, 0(r2)`
- `sd f6, 8(r2)`
- `sd f7, 16(r2)`
- `sd f8, 24(r2)`
- `add r2, 32`
- `bne r1, r3, 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>loop:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>ld f1</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>ld f2</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>ld f3</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>fadd f5</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>fadd f6</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>fadd f7</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>fadd f8</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>sd f5</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>sd f6</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>sd f7</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>bne r1, r3, loop</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>sd f8</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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>loop:</td>
<td>ld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r1, 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f5, f0, f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f6, f0, f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f7, f0, f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f8, f0, f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sd f5, 0(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sd f6, 8(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sd f7, 16(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sd f8, 24(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r2, 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne r1, r3, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Schedule

How many FLOPS/cycle?

April 23, 2020

MIT 6.823 Spring 2020
Scheduling Loop Unrolled Code

Unroll 4 ways

```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, loop
```

Schedule

```
Int1 | Int 2 | M1 | M2 | FP+ | FPx
--- | --- | --- | --- | --- | ---
ld f1 |   |   |   |     |     
ld f2 |   |   |     |     |     
ld f3 |   |     |     |     |     
ld f4 | add r1 |   |     |     |     
fadd f5 |   |     |     |     |     
fadd f6 |   |     |     |     |     
fadd f7 |   |     |     |     |     
fadd f8 | sd f5 |  |     |     |     
sd f6 | sd f6 |  |     |     |     
sd f7 | sd f7 |  |     |     |     
sd f8 | add r2 | bne |     |     |     
```

How many FLOPS/cycle?

4 fadds / 11 cycles = 0.36
## Software Pipelining

**Unroll 4 ways first**

```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       add r2, 32
       sd f8, -8(r2)
       bne r1, r3, loop
```

<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></td>
<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>
<td></td>
</tr>
<tr>
<td></td>
<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>
<td></td>
</tr>
<tr>
<td></td>
<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>
<td></td>
</tr>
<tr>
<td></td>
<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>
<td></td>
</tr>
<tr>
<td></td>
<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>
<td></td>
</tr>
<tr>
<td></td>
<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>
<td></td>
</tr>
<tr>
<td></td>
<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>
<td></td>
</tr>
<tr>
<td></td>
<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>
<td></td>
</tr>
<tr>
<td></td>
<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>
<td></td>
</tr>
<tr>
<td></td>
<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>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Software Pipelining

## Unroll 4 ways first

```python
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       add r2, 32
       sd f8, -8(r2)
       bne r1, r3, loop
```

<table>
<thead>
<tr>
<th></th>
<th>Int1</th>
<th>Int2</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>
<td></td>
</tr>
<tr>
<td>Int 1</td>
<td>ld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Int 2</td>
<td></td>
<td>ld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>add r1</td>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fadd f5</td>
<td></td>
<td>FP+</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fadd f6</td>
<td></td>
<td></td>
<td>FPx</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fadd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fadd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>add r2</td>
<td>sd f7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bne</td>
<td></td>
<td>sd f8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<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>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
## Software Pipelining

**Unroll 4 ways first**

```
loop:  ld f1, 0(r1)  
    ld f2, 8(r1)  
    ld f3, 16(r1)  
    ld f4, 24(r1)  
    add r1, 32  
    fadd f5, f0, f1  
    fadd f6, f0, f2  
    fadd f7, f0, f3  
    fadd f8, f0, f4  
    sd f5, 0(r2)  
    sd f6, 8(r2)  
    sd f7, 16(r2)  
    add r2, 32  
    sd f8, -8(r2)  
    bne r1, r3, loop
```

<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>ld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
<td>fadd f5</td>
<td></td>
</tr>
<tr>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
<td>fadd f6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fadd f7</td>
<td></td>
</tr>
<tr>
<td>add r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f7</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>add r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</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>
<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>
# 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:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f1, 0(r1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f2, 8(r1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f3, 16(r1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f4, 24(r1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1</td>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1, 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f5, f0, f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f6, f0, f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f7, f0, f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f8, f0, f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f5, 0(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f6, 8(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f7, 16(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2, 32</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>
</tbody>
</table>

April 23, 2020

MIT 6.823 Spring 2020
# Software Pipelining

**Unroll 4 ways first**

```plaintext
loop:  ld f1, 0(r1)
      ld f2, 8(r1)
      ld f3, 16(r1)
      ld f4, 24(r1)
      add r1, 32
      fadd f5, f0, f1
      fadd f6, f0, f2
      fadd f7, f0, f3
      fadd f8, f0, f4
      sd f5, 0(r2)
      sd f6, 8(r2)
      sd f7, 16(r2)
      add r2, 32
      sd f8, -8(r2)
      bne r1, r3, loop
```

<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>ld f1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld f2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld f3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>add r1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld f1</td>
<td></td>
<td>fadd f5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld f2</td>
<td></td>
<td>fadd f6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld f3</td>
<td></td>
<td>fadd f7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>add r1</td>
<td></td>
<td>fadd f8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld f1</td>
<td></td>
<td>sd f5</td>
<td>fadd f5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld f2</td>
<td></td>
<td>sd f6</td>
<td>fadd f6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add r2</td>
<td></td>
<td>sd f7</td>
<td>fadd f7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>add r1</td>
<td></td>
<td>bne</td>
<td>ld f4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bne</td>
<td></td>
<td>sd f8</td>
<td>fadd f8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>add r2</td>
<td></td>
<td>sd f7</td>
<td>fadd f7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bne</td>
<td></td>
<td>sd f8</td>
<td>fadd f8</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>sd f5</td>
<td></td>
</tr>
</tbody>
</table>
```

April 23, 2020

MIT 6.823 Spring 2020
### Software Pipelining

#### Unroll 4 ways first

**Loop**
- ld f1, 0(r1)
- ld f2, 8(r1)
- ld f3, 16(r1)
- ld f4, 24(r1)
- add r1, 32
- fadd f5, f0, f1
- fadd f6, f0, f2
- fadd f7, f0, f3
- fadd f8, f0, f4
- sd f5, 0(r2)
- sd f6, 8(r2)
- sd f7, 16(r2)
- add r2, 32
- sd f8, -8(r2)
- bne r1, r3, loop

**Prolog**
- ld f1
- ld f2
- ld f3
- ld f4
- add r1

**Iterate**
- ld f1
- ld f2
- ld f3
- ld f4
- add r2
- add r1, bne

**Epilog**
- ld f5
- ld f6
- ld f7
- ld f8

### 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>ld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

April 23, 2020
How many FLOPS/cycle?

4 fadds / 4 cycles = 1
Software Pipelining vs. Unrolling

**Loop Unrolling**
- **Startup overhead**
- **Wind-down overhead**

**Software Pipelining**
- **Startup overhead**

**Performance**
- Loop Iteration
- Time

**Software pipelining pays startup/wind-down costs only once per loop, not once per iteration**

April 23, 2020
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
Trace Scheduling
[Fisher, Ellis]

- Pick string of basic blocks, a trace, that represents most frequent branch path
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace
Trace Scheduling
[Fisher, Ellis]

- Pick string of basic blocks, a trace, that represents most frequent branch path
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace
Trace Scheduling [Fisher, Ellis]

- Pick string of basic blocks, a trace, that represents most frequent branch path
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace

How do we know which trace to pick?
Trace Scheduling
[Fisher, Ellis]

- Pick string of basic blocks, a trace, that represents most frequent branch path
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace

How do we know which trace to pick?
Use profiling feedback or compiler heuristics to find common branch paths
Problems with “Classic” VLIW
Problems with “Classic” VLIW

- Knowing branch probabilities
  - Profiling requires an significant extra step in build process
Problems with “Classic” VLIW

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

• Scheduling for statically unpredictable branches
  – Optimal schedule varies with branch path
Problems with “Classic” VLIW

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

- Scheduling for statically unpredictable branches
  - Optimal schedule varies with branch path

- Object code size
  - Instruction padding wastes instruction memory/cache
  - Loop unrolling/software pipelining replicates code
Problems with “Classic” VLIW

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

- Scheduling for statically unpredictable branches
  - Optimal schedule varies with branch path

- Object code size
  - Instruction padding wastes instruction memory/cache
  - Loop unrolling/software pipelining replicates code

- Scheduling memory operations
  - Caches and/or memory bank conflicts impose statically unpredictable variability
  - Uncertainty about addresses limit code reordering
Problems with “Classic” VLIW

- Knowing branch probabilities
  - Profiling requires an significant extra step in build process
- Scheduling for statically unpredictable branches
  - Optimal schedule varies with branch path
- Object code size
  - Instruction padding wastes instruction memory/cache
  - Loop unrolling/software pipelining replicates code
- Scheduling memory operations
  - Caches and/or memory bank conflicts impose statically unpredictable variability
  - Uncertainty about addresses limit code reordering
- Object-code compatibility
  - Have to recompile all code for every machine, even for two machines in same generation
### 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
  - Provide a single-op VLIW instruction
    - Cydra-5 UniOp instructions
  - Mark parallel groups
    - used in TMS320C6x DSPs, Intel IA-64

![VLIW Instruction Diagram](attachment:image.png)
Cydra-5: Memory Latency Register (MLR)
Cydra-5: Memory Latency Register (MLR)

- Problem: Loads have variable latency
Cydra-5: Memory Latency Register (MLR)

- Problem: Loads have variable latency
- Solution: Let software choose desired memory latency
Cydra-5: Memory Latency Register (MLR)

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

• Compiler schedules code for maximum load-use distance
Cydra-5: 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
Cydra-5: 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
IA-64 Predicated Execution
IA-64 Predicated Execution

Problem: Mispredicted branches limit ILP
IA-64 Predicated Execution

Problem: Mispredicted branches limit ILP
Solution: Eliminate hard-to-predict branches with predicated execution
  - Almost all IA-64 instructions can be executed conditionally under predicate
  - Instruction becomes NOP if predicate register false
Problem: Mispredicted branches limit ILP
Solution: Eliminate hard-to-predict branches with predicated execution
- Almost all IA-64 instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false

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

b1:  Inst 3  else
     Inst 4
     br b3

b2:  Inst 5  then
     Inst 6

b3:  Inst 7
     Inst 8
```

Four basic blocks
**IA-64 Predicated Execution**

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

- Almost all IA-64 instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false

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

Inst 3 else
Inst 4
br b3

Inst 5 then
Inst 6

Inst 7
Inst 8
```

**Predication**

```
Inst 1
Inst 2
p1,p2 ← cmp(a==b)
(p1) Inst 3 || (p2) Inst 5
(p1) Inst 4 || (p2) Inst 6
Inst 7
Inst 8
```

**One basic block**

*Mahlke et al, ISCA95: On average >50% branches removed*
Where does predication fit in?
IA-64 Speculative Execution

Problem: Branches restrict compiler code motion

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

Load r1
Use r1
Inst 3
```
Problem: Branches restrict compiler code motion

Can't move load above branch because might cause spurious exception
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
IA-64 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

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

Chk.s r1
Use r1
Inst 3

Check for exception in original home block jumps to fixup code if exception detected
```
IA-64 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

Particularly useful for scheduling long latency loads early
Problem: Possible memory hazards limit code scheduling
IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

Can't move load above store because store might be to same address
IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Instruction-based speculation with hardware monitor to check for pointer hazards

Can’t move load above store because store might be to same address
IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Instruction-based speculation with hardware monitor to check for pointer hazards

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

Data speculative load adds address to address check table

Store invalidates any matching loads in address check table

Check if load invalid (or missing), jump to fixup code if so
IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

Solution: Instruction-based speculation with hardware monitor to check for pointer hazards

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

Data speculative load adds address to address check table

Store invalidates any matching loads in address check table

Check if load invalid (or missing), jump to fixup code if so

Requires associative hardware in address check table
Clustered VLIW

- Divide machine into clusters of local register files and local functional units
- Lower bandwidth/higher latency interconnect between clusters
- Software responsible for mapping computations to minimize communication overhead
- Common in commercial embedded processors, examples include TI C6x series DSPs, and HP Lx processor
- Exists in some superscalar processors, e.g., Alpha 21264
Limits of Static Scheduling

- Unpredictable branches
- Unpredictable memory behavior (cache misses and dependencies)
- Code size explosion
- Compiler complexity
Limits of Static Scheduling

- Unpredictable branches
- Unpredictable memory behavior (cache misses and dependencies)
- Code size explosion
- Compiler complexity

Question:

How applicable are the VLIW-inspired techniques to traditional RISC/CISC processor architectures?
Thank you!

Next Lecture: Vector Processors