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”)

November 6, 2023  MIT 6.5900 Fall 2023  L17-2
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
Microcoded Microarchitecture

μcontroller (ROM)

Memory (RAM)

Datapath

busy?
zero?
opcode

enMem
MemWrt

Data
Addr

holds fixed microcode instructions
Microcoded Microarchitecture

- **µcontroller (ROM)**: Holds fixed microcode instructions.
- **Memory (RAM)**: Holds user program written in macrocode instructions (e.g., MIPS, x86, RISC-V, etc.).
- **Datapath**: Interfaces with other components through signals like `Data`, `Addr`, `enMem`, `MemWrt`.
- **Data Path Inputs**: `busy?`, `zero?`, `opcode`.

Signal Definitions:
- `enMem`: Enables Memory operations.
- `MemWrt`: Enables Memory write operations.

Diagram illustrates the flow of data and control signals between these components.
A Bus-based Datapath for RISC-V
A Bus-based Datapath for RISC-V

Diagram showing the components of a bus-based datapath with labels for Opcode, ldIR, IdIR, OpSel, ldA, ldB, zero?, ImmSel, Imm Gen, Imm Gen control, ALU, ALU control, enImm, enALU, and Bus 32.
A Bus-based Datapath for RISC-V
A Bus-based Datapath for RISC-V
A Bus-based Datapath for RISC-V

Microinstruction: register to register transfer (17 control signals)

MA ← PC  means  RegSel = PC;  enReg=yes;  IdMA= yes
B ← Reg[rs1]  means
A Bus-based Datapath for RISC-V

Microinstruction: register to register transfer (17 control signals)

MA ← PC means RegSel = PC; enReg=yes; IdMA=yes
B ← Reg[rs1] means RegSel = rs1; enReg=yes; IdB=yes
• Assumption: Memory operates asynchronously and is slow compared to Reg-to-Reg transfers
Microcode Controller

Opcode → \( ext \)

absolute

\( \mu PC \) (state)

\( +1 \)

\( \mu PC + 1 \)

Control ROM

address

data

zero

busy

Control Signals (17)
Microcode Controller

Control Signals (17)

Input encoding reduces ROM height
Microcode Controller

Control Signals (17)

 Opcode → ext

 absolute

 op-group

 μPC (state)

 +1

 μPCSrc

 jump logic

 Control ROM

 address

 data

 Control Signals (17)

 next-state encoding reduces ROM width

 input encoding reduces ROM height

 busy

 zero

 μPC

 μPC+1

 L17-7
Microcode Controller

- Microcode Controller
- Opcode: ext
- absolute
- op-group
- \( \mu PC \) \( \mu PC+1 \)
- +1
- \( \mu PC \text{Src} \)
- jump logic
- jump
- zero
- busy
- \( \mu \text{JumpType} = \) next | spin | fetch | dispatch | feqz | fnez
- Control ROM
- address
- data
- Input encoding reduces ROM height
- Next-state encoding reduces ROM width
- Control Signals (17)
Jump Logic

\[ \mu\text{PCSrc} = \text{Case} \quad \mu\text{JumpTypes} \]

- next \( \Rightarrow \) \( \mu\text{PC} + 1 \)
- spin \( \Rightarrow \) if (busy) then \( \mu\text{PC} \) else \( \mu\text{PC} + 1 \)
- fetch \( \Rightarrow \) absolute
- dispatch \( \Rightarrow \) op-group
- feqz \( \Rightarrow \) if (zero) then absolute else \( \mu\text{PC} + 1 \)
- fnez \( \Rightarrow \) if (zero) then \( \mu\text{PC} + 1 \) else absolute
Execution of a RISC-V 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>fetch₀</td>
<td>MA ← PC</td>
<td></td>
</tr>
<tr>
<td>fetch₁</td>
<td>IR ← Memory</td>
<td></td>
</tr>
<tr>
<td>fetch₂</td>
<td>A ← PC</td>
<td></td>
</tr>
<tr>
<td>fetch₃</td>
<td>PC ← A + 4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU₀</td>
<td>A ← Reg[rs₁]</td>
<td></td>
</tr>
<tr>
<td>ALU₁</td>
<td>B ← Reg[rs₂]</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 ← sExt(Imm)</td>
<td></td>
</tr>
<tr>
<td>ALUᵢ₂</td>
<td>Reg[rd] ← Op(A,B)</td>
<td></td>
</tr>
</tbody>
</table>
Instruction Fetch

State | Control points | next-state
--- | --- | ---
fetch₀ | MA ← PC | next
fetch₁ | IR ← Memory | 
fetch₂ | A ← PC | 
fetch₃ | PC ← A + 4 | 
... | 
ALU₀ | A ← Reg[rs₁] | 
ALU₁ | B ← Reg[rs₂] | 
ALU₂ | Reg[rd] ← func(A,B) | 
ALUᵢ₀ | A ← Reg[rs] | 
ALUᵢ₁ | B ← sExt(Imm) | 
ALUᵢ₂ | 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></td>
</tr>
<tr>
<td>fetch₃</td>
<td>PC ← A + 4</td>
<td></td>
</tr>
</tbody>
</table>

... 

ALU₀  A ← Reg[rs₁]
ALU₁  B ← Reg[rs₂]
ALU₂  Reg[rd] ← func(A, B)

ALUᵢ₀ A ← Reg[rs]
ALUᵢ₁ B ← sExt(Imm)
ALUᵢ₂ 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></td>
</tr>
</tbody>
</table>

...  

ALU₀   | A ← Reg[rs₁]            |            |
ALU₁   | B ← Reg[rs₂]            |            |
ALU₂   | Reg[rd] ← func(A,B)     |            |

ALUi₀  | A ← Reg[rs]             |            |
ALUi₁  | B ← sExt(Imm)           |            |
ALUi₂  | Reg[rd] ← Op(A,B)       |            |

---

![Instruction Fetch Diagram](image-url)
Instruction Fetch

State | Control points | next-state
---|---|---
fetch\(_0\) | MA ← PC | next
fetch\(_1\) | IR ← Memory | spin
fetch\(_2\) | A ← PC | next
fetch\(_3\) | PC ← A + 4 | dispatch
...

ALU\(_0\) | A ← Reg[rs1] |
ALU\(_1\) | B ← Reg[rs2] |
ALU\(_2\) | Reg[rd] ← func(A,B) |

ALUi\(_0\) | A ← Reg[rs] |
ALUi\(_1\) | B ← sExt(Imm) |
ALUi\(_2\) | 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>
</tbody>
</table>

...  

ALU₀  
ALU₁  
ALU₂  

ALUi₀  
ALUi₁  
ALUi₂  

![Diagram](image-url)
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[rs₁]</td>
<td>next</td>
</tr>
<tr>
<td>ALU₁</td>
<td>B ← Reg[rs₂]</td>
<td>next</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 ← sExt(Imm)</td>
<td></td>
</tr>
<tr>
<td>ALUᵢ₂</td>
<td>Reg[rd]← Op(A,B)</td>
<td></td>
</tr>
</tbody>
</table>

[Diagram of the Instruction Fetch process]
## 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>next</td>
</tr>
<tr>
<td>fetch(_3)</td>
<td>PC ← A + 4</td>
<td>dispatch</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU(_0)</td>
<td>A ← Reg[rs1]</td>
<td>next</td>
</tr>
<tr>
<td>ALU(_1)</td>
<td>B ← Reg[rs2]</td>
<td>next</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>next</td>
</tr>
<tr>
<td>ALU(_i_1)</td>
<td>B ← sExt(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 Diagram](image-url)
Instruction Fetch

State | Control points | next-state
--- | --- | ---
fetch<sub>0</sub> | MA ← PC | next
fetch<sub>1</sub> | IR ← Memory | spin
fetch<sub>2</sub> | A ← PC | next
fetch<sub>3</sub> | PC ← A + 4 | dispatch
... | | |
ALU<sub>0</sub> | A ← Reg[rs1] | next
ALU<sub>1</sub> | B ← Reg[rs2] | next
ALU<sub>2</sub> | Reg[rd] ← func(A,B) | fetch
ALUi<sub>0</sub> | A ← Reg[rs] | next
ALUi<sub>1</sub> | B ← sExt(Imm) | next
ALUi<sub>2</sub> | Reg[rd] ← Op(A,B) | fetch

![Diagram of Instruction Fetch](image-url)
## 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[rs1]</td>
<td>next</td>
</tr>
<tr>
<td>LW₁</td>
<td>B ← sExt(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[rd] ← Memory</td>
<td>spin</td>
</tr>
<tr>
<td>LW₄</td>
<td>fetch</td>
<td></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[rs1]</td>
<td>next</td>
</tr>
<tr>
<td>LW₁</td>
<td>B ← sExt(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[rd] ← Memory</td>
<td>spin</td>
</tr>
<tr>
<td>LW₄</td>
<td></td>
<td>fetch</td>
</tr>
<tr>
<td>SW₀</td>
<td>A ← Reg[rs1]</td>
<td>next</td>
</tr>
<tr>
<td>SW₁</td>
<td>B ← sExt(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[rs2]</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>BEQ₀</td>
<td>A ← Reg[rs1]</td>
<td>next</td>
</tr>
<tr>
<td>BEQ₁</td>
<td>B ← Reg[rs2]</td>
<td>next</td>
</tr>
<tr>
<td>BEQ₂</td>
<td>A ← A - B</td>
<td>next</td>
</tr>
<tr>
<td>BEQ₃</td>
<td></td>
<td>fnez</td>
</tr>
<tr>
<td>BEQ₄</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>BEQ₅</td>
<td>B ← sExt(Imm&lt;&lt;1)</td>
<td>next</td>
</tr>
<tr>
<td>BEQ₆</td>
<td>PC ← A+B</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>BEQ₀</td>
<td>$A \leftarrow \text{Reg[rs1]}$</td>
<td>next</td>
</tr>
<tr>
<td>BEQ₁</td>
<td>$B \leftarrow \text{Reg[rs2]}$</td>
<td>next</td>
</tr>
<tr>
<td>BEQ₂</td>
<td>$A \leftarrow A - B$</td>
<td>next</td>
</tr>
<tr>
<td>BEQ₃</td>
<td></td>
<td>fnez</td>
</tr>
<tr>
<td>BEQ₄</td>
<td>$A \leftarrow \text{PC}$</td>
<td>next</td>
</tr>
<tr>
<td>BEQ₅</td>
<td>$B \leftarrow \text{sExt(Imm&lt;&lt;1)}$</td>
<td>next</td>
</tr>
<tr>
<td>BEQ₆</td>
<td>$\text{PC} \leftarrow A+B$</td>
<td>fetch</td>
</tr>
<tr>
<td>BNE₀</td>
<td>$A \leftarrow \text{Reg[rs1]}$</td>
<td>next</td>
</tr>
<tr>
<td>BNE₁</td>
<td>$B \leftarrow \text{Reg[rs2]}$</td>
<td>next</td>
</tr>
<tr>
<td>BNE₂</td>
<td>$A \leftarrow A - B$</td>
<td>next</td>
</tr>
<tr>
<td>BNE₃</td>
<td></td>
<td>feqz</td>
</tr>
<tr>
<td>BNE₄</td>
<td>$A \leftarrow \text{PC}$</td>
<td>next</td>
</tr>
<tr>
<td>BNE₅</td>
<td>$B \leftarrow \text{sExt(Imm&lt;&lt;1)}$</td>
<td>next</td>
</tr>
<tr>
<td>BNE₆</td>
<td>$\text{PC} \leftarrow A+B$</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>BLT₀</td>
<td>A ← Reg[rs1]</td>
<td>next</td>
</tr>
<tr>
<td>BLT₁</td>
<td>B ← Reg[rs2]</td>
<td>next</td>
</tr>
<tr>
<td>BLT₂</td>
<td>A ← slt(A, B)</td>
<td>next</td>
</tr>
<tr>
<td>BLT₃</td>
<td></td>
<td>feqz</td>
</tr>
<tr>
<td>BLT₄</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>BLT₅</td>
<td>B ← sExt(Imm&lt;&lt;1)</td>
<td>next</td>
</tr>
<tr>
<td>BLT₆</td>
<td>PC ← A+B</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>BLT₀</td>
<td>A ← Reg[rs1]</td>
<td>next</td>
</tr>
<tr>
<td>BLT₁</td>
<td>B ← Reg[rs2]</td>
<td>next</td>
</tr>
<tr>
<td>BLT₂</td>
<td>A ← slt(A, B)</td>
<td>next</td>
</tr>
<tr>
<td>BLT₃</td>
<td></td>
<td>feqz</td>
</tr>
<tr>
<td>BLT₄</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>BLT₅</td>
<td>B ← sExt(Imm&lt;&lt;1)</td>
<td>next</td>
</tr>
<tr>
<td>BLT₆</td>
<td>PC ← A+B</td>
<td>fetch</td>
</tr>
</tbody>
</table>

Similar sequences for BGE, BLTU, BGEU
## Jumps

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL₀</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>JAL₁</td>
<td>Reg[rd] ← A</td>
<td>next</td>
</tr>
<tr>
<td>JAL₂</td>
<td>B ← IR</td>
<td>next</td>
</tr>
<tr>
<td>JAL₃</td>
<td>PC ← JumpTarg(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>JALR₀</td>
<td>A ← Reg[rs1]</td>
<td>next</td>
</tr>
<tr>
<td>JALR₁</td>
<td>B ← PC</td>
<td>next</td>
</tr>
<tr>
<td>JALR₂</td>
<td>Reg[rd] ← B</td>
<td>next</td>
</tr>
<tr>
<td>JALR₃</td>
<td>B ← IR</td>
<td>next</td>
</tr>
<tr>
<td>JALR₄</td>
<td>PC ← JumpTarg(A,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

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

Find independent operations
Sequential ISA Bottleneck

Sequential source code

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

Superscalar compiler

Find independent operations

Schedule operations
Sequential ISA Bottleneck

Sequential source code

$ a = \text{foo}(b); \
\text{for } (i=0, i< 

Superscalar compiler

Find independent operations

Schedule operations

Sequential machine code

November 6, 2023

MIT 6.5900 Fall 2023
Sequential ISA Bottleneck

Sequential source code

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

Superscalar compiler

Find independent operations

Schedule operations

Sequential machine code

Superscalar processor

Check instruction dependencies
Sequential ISA Bottleneck

Sequential source code

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

Sequential machine code

Superscalar compiler

- Find independent operations
- Schedule operations

Superscalar processor

- Check instruction dependencies
- Schedule execution
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

Diagram:

|----------|----------|----------|----------|---------|---------|

- 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
- 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++)

Loop Execution

Compile:

```
loop:  fld f1, 0(x1)
       add x1, 8
       fadd.d f2, f0, f1
       fsd f2, 0(x2)
       add x2, 8
       bne x1, x3, loop
```

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

Compile

loop: fld f1, 0(x1)
    add x1, 8
    fadd.d f2, f0, f1
    fsd f2, 0(x2)
    add x2, 8
    bne x1, x3, loop
### Loop Execution

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

#### Compile

```
loop:  fld f1, 0(x1)
    add x1, 8
    fadd.d f2, f0, f1
    fsd f2, 0(x2)
    add x2, 8
    bne x1, x3, 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 x1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld</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>bne x1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne x3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
for (i=0; i<N; i++)

Compile

loop:  fld f1, 0(x1)
       add x1, 8
       fadd.d f2, f0, f1
       fsd f2, 0(x2)
       add x2, 8
       bne x1, x3, loop

Schedule

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>add x1</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>fadd.d</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>
for (i=0; i<N; i++)

Compile

loop:  fld f1, 0(x1)
       add x1, 8
       fadd.d f2, f0, f1
       fsd f2, 0(x2)
       add x2, 8
       bne x1, x3, 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>fadd.d</td>
<td></td>
</tr>
</tbody>
</table>

loop:
Loop Execution

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

Compile

loop:  fld f1, 0(x1)
       add x1, 8
       fadd.d f2, f0, f1
       fsd f2, 0(x2)
       add x2, 8
       bne x1, x3, 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 x1</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>fld</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>fadd.d</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>fsd</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>

<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>loop:</td>
<td>add x1</td>
<td>fld</td>
<td>fadd.d</td>
<td>fsd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
for (i=0; i<N; i++)

loop:
    fld f1, 0(x1)
    add x1, 8
    fadd.d f2, f0, f1
    fsd f2, 0(x2)
    add x2, 8
    bne x1, x3, loop
### Loop Execution

#### Compile

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

#### Schedule

```
loop:  fld f1, 0(x1)
       add x1, 8
       fadd.d f2, f0, f1
       fsd f2, 0(x2)
       add x2, 8
       bne x1, x3, 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>add x1</td>
<td>fld</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd.d</td>
<td>f2</td>
<td>f0</td>
<td>f1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd</td>
<td>f2, 0(x2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add x2</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne x1, x3, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Loop Execution

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

**Compile**

```
loop:  fld f1, 0(x1)
      add x1, 8
      fadd.d f2, f0, f1
      fsd f2, 0(x2)
      add x2, 8
      bne x1, x3, 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 x1</td>
<td>fld</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add x2</td>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fsd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Loop Execution

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

Compile

loop:  fld f1, 0(x1)
       add x1, 8
       fadd.d f2, f0, f1
       fsd f2, 0(x2)
       add x2, 8
       bne x1, x3, loop

Schedule

How many FP ops/cycle?
Loop Execution

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

Compile

loop:  fld f1, 0(x1)
       add x1, 8
       fadd.d f2, f0, f1
       fsd f2, 0(x2)
       add x2, 8
       bne x1, x3, loop

Schedule

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>add x1</td>
<td>fld</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fadd.d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add x2</td>
<td>bne</td>
<td>fsd</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Unroll inner loop to perform 4 iterations at once

Original:

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

Unrolled:

```c
for (i=0; i<N; i+=4)
{
}
```
Loop Unrolling

Unroll inner loop to perform 4 iterations at once

Is this code always correct?
Loop Unrolling

Unroll inner loop to perform 4 iterations at once

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

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

Is this code always 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**

<table>
<thead>
<tr>
<th>loop:</th>
<th>fld f1, 0(x1)</th>
<th>fld f2, 8(x1)</th>
<th>fld f3, 16(x1)</th>
<th>fld f4, 24(x1)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add x1, 32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f5, f0, f1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f6, f0, f2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f7, f0, f3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f8, f0, f4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f5, 0(x2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f6, 8(x2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f7, 16(x2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f8, 24(x2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add x2, 32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne x1, x3, loop</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Schedule**
### Scheduling Loop Unrolled Code

#### Unroll 4 ways

- loop: `fld f1, 0(x1)`
- `fld f2, 8(x1)`
- `fld f3, 16(x1)`
- `fld f4, 24(x1)`
- `add x1, 32`
- `fadd.d f5, f0, f1`
- `fadd.d f6, f0, f2`
- `fadd.d f7, f0, f3`
- `fadd.d f8, f0, f4`
- `fsd f5, 0(x2)`
- `fsd f6, 8(x2)`
- `fsd f7, 16(x2)`
- `fsd f8, 24(x2)`
- `add x2, 32`
- `bne x1, x3, 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></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>

**L17-23**
Scheduling Loop Unrolled Code

Unroll 4 ways

loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd.d f5, f0, f1
       fadd.d f6, f0, f2
       fadd.d f7, f0, f3
       fadd.d f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       fsd f8, 24(x2)
       add x2, 32
       bne x1, x3, loop
### Unroll 4 ways

Unroll the loop 4 ways:

```assembly
loop:  fld f1, 0(x1)
      fld f2, 8(x1)
      fld f3, 16(x1)
      fld f4, 24(x1)
      add x1, 32
      fadd.d f5, f0, f1
      fadd.d f6, f0, f2
      fadd.d f7, f0, f3
      fadd.d f8, f0, f4
      fsd f5, 0(x2)
      fsd f6, 8(x2)
      fsd f7, 16(x2)
      fsd f8, 24(x2)
      add x2, 32
      bne x1, x3, loop
```

### Schedule

#### Int 1
- Field f1

#### Int 2
- Field f2
- Field f3

#### M1
- Field f4

#### M2

#### FP+

#### FPx

---

**L17-23**
Scheduling Loop Unrolled Code

Unroll 4 ways

| loop: | fld f1, 0(x1) | fld f2, 8(x1) | fld f3, 16(x1) | fld f4, 24(x1) | add x1, 32 | fadd.d f5, f0, f1 | fadd.d f6, f0, f2 | fadd.d f7, f0, f3 | fadd.d f8, f0, f4 | fsd f5, 0(x2) | fsd f6, 8(x2) | fsd f7, 16(x2) | fsd f8, 24(x2) | add x2, 32 | bne x1, x3, loop |

Schedule

<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>loop:</td>
<td>fld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fld f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add x1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fld f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add x2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne x1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x3, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scheduling Loop Unrolled Code

**Unroll 4 ways**

<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>fld f1, 0(x1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f2, 8(x1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f3, 16(x1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f4, 24(x1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add x1, 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd.d f5, f0, f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd.d f6, f0, f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd.d f7, f0, f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd.d f8, f0, f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd f5, 0(x2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd f6, 8(x2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd f7, 16(x2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd f8, 24(x2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add x2, 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne x1, x3, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Schedule**
Scheduling Loop Unrolled Code

Unroll 4 ways

loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd.d f5, f0, f1
       fadd.d f6, f0, f2
       fadd.d f7, f0, f3
       fadd.d f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       fsd f8, 24(x2)
       add x2, 32
       bne x1, x3, 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></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>Int1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Int 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scheduling Loop Unrolled Code

**Unroll 4 ways**

```
loop:  fld f1, 0(x1)  
      fld f2, 8(x1)  
      fld f3, 16(x1)  
      fld f4, 24(x1)  
      add x1, 32  
      fadd.d f5, f0, f1  
      fadd.d f6, f0, f2  
      fadd.d f7, f0, f3  
      fadd.d f8, f0, f4  
      fsd f5, 0(x2)  
      fsd f6, 8(x2)  
      fsd f7, 16(x2)  
      fsd f8, 24(x2)  
      add x2, 32  
      bne x1, x3, loop
```
Scheduling Loop Unrolled Code

Unroll 4 ways

```c
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd.d f5, f0, f1
       fadd.d f6, f0, f2
       fadd.d f7, f0, f3
       fadd.d f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       fsd f8, 24(x2)
       add x2, 32
       bne x1, x3, 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>fld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd.d f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add x2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Scheduling Loop Unrolled Code

Unroll 4 ways

<table>
<thead>
<tr>
<th>loop: fld f1, 0(x1)</th>
<th>fld f2, 0(x1)</th>
<th>fld f3, 0(x1)</th>
<th>fld f4, 0(x1)</th>
<th>add x1, 32</th>
<th>fadd.d f5, f0, f1</th>
<th>fadd.d f6, f0, f2</th>
<th>fadd.d f7, f0, f3</th>
<th>fadd.d f8, f0, f4</th>
<th>fsd f5, 0(x2)</th>
<th>fsd f6, 0(x2)</th>
<th>fsd f7, 0(x2)</th>
<th>fsd f8, 0(x2)</th>
<th>add x2, 32</th>
</tr>
</thead>
</table>

Schedule

How many FLOPS/cycle?
Scheduling Loop Unrolled Code

Unroll 4 ways

How many FLOPS/cycle?

4 fadds / 11 cycles = 0.36
## Software Pipelining

### Unroll 4 ways first

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd.d f5, f0, f1
       fadd.d f6, f0, f2
       fadd.d f7, f0, f3
       fadd.d f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       add x2, 32
       fsd f8, -8(x2)
       bne x1, x3, loop
```
# Software Pipelining

## Unroll 4 ways first

<table>
<thead>
<tr>
<th>loop: fld f1, 0(x1)</th>
<th>fld f1</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td>fld f2, 8(x1)</td>
<td>fld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f3, 16(x1)</td>
<td>fld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f4, 24(x1)</td>
<td>add x1 f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add x1, 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd.d f5, f0, f1</td>
<td>fadd.d f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd.d f6, f0, f2</td>
<td>fadd.d f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd.d f7, f0, f3</td>
<td>fadd.d f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd.d f8, f0, f4</td>
<td>fadd.d f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd f5, 0(x2)</td>
<td>fsd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd f6, 8(x2)</td>
<td>fsd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd f7, 16(x2)</td>
<td>add x2 f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add x2, 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd f8, -8(x2)</td>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne x1, x3, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Software Pipelining

Unroll 4 ways first

loop: fld f1, 0(x1)
   fld f2, 8(x1)
   fld f3, 16(x1)
   fld f4, 24(x1)
   add x1, 32
   fadd.d f5, f0, f1
   fadd.d f6, f0, f2
   fadd.d f7, f0, f3
   fadd.d f8, f0, f4
   fsd f5, 0(x2)
   fsd f6, 8(x2)
   fsd f7, 16(x2)
   add x2, 32
   fsd f8, -8(x2)
   bne x1, x3, loop

Int1 | Int 2 | M1 | M2 | FP+ | FPx
--- | --- | --- | --- | --- | ---
    |    | fld f1 |    |    |    
    |    | fld f2 |    |    |    
    |    | fld f3 |    |    |    
    |    | fld f4 |    |    |    
    | add x1 | fld f5 |    | fadd.d f5 |    
    |    | fld f6 |    | fadd.d f6 |    
    |    | fld f7 |    | fadd.d f7 |    
    | add x1 | fld f8 |    | fadd.d f8 |    
    |    | fsd f5 |    | fadd.d f5 |    
    |    | fsd f6 |    | fadd.d f6 |    
    | add x2 | fsd f7 |    | fadd.d f7 |    
    |    | bne |    | fsd f8 | fadd.d f8 
    |    | fsd f5 |    |    |    
    |    | fsd f6 |    |    |    
    | add x2 | fsd f7 |    |    |    
    |    | bne |    | fsd f8 |    

L17-24
## 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>fld f1</td>
<td></td>
<td></td>
<td>fld f1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f2</td>
<td></td>
<td></td>
<td>fld f2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f3</td>
<td></td>
<td></td>
<td>fadd.d f5</td>
<td>fadd.d f5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f4</td>
<td></td>
<td></td>
<td>fadd.d f6</td>
<td>fadd.d f6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd.d f5, f0, f1</td>
<td></td>
<td></td>
<td>fadd.d f7</td>
<td>fadd.d f7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd.d f6, f0, f2</td>
<td></td>
<td></td>
<td>fadd.d f8</td>
<td>fadd.d f8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd.d f7, f0, f3</td>
<td></td>
<td></td>
<td>fld f1</td>
<td>fadd.d f5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd.d f8, f0, f4</td>
<td></td>
<td></td>
<td>fld f2</td>
<td>fadd.d f6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd f5, 0(x2)</td>
<td></td>
<td></td>
<td>fld f3</td>
<td>fadd.d f7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd f6, 8(x2)</td>
<td></td>
<td></td>
<td>fld f4</td>
<td>fadd.d f8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd f7, 16(x2)</td>
<td></td>
<td></td>
<td>fadd.d f15</td>
<td>fadd.d f15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add x2</td>
<td></td>
<td></td>
<td>fadd.d f16</td>
<td>fadd.d f16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add x2</td>
<td></td>
<td></td>
<td>fadd.d f17</td>
<td>fadd.d f17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add x1</td>
<td></td>
<td></td>
<td>fadd.d f18</td>
<td>fadd.d f18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f1</td>
<td></td>
<td></td>
<td>fadd.d f5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f2</td>
<td></td>
<td></td>
<td>fadd.d f6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f3</td>
<td></td>
<td></td>
<td>fadd.d f7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f4</td>
<td></td>
<td></td>
<td>fadd.d f8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f5</td>
<td></td>
<td></td>
<td>fadd.d f15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f6</td>
<td></td>
<td></td>
<td>fadd.d f16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f7</td>
<td></td>
<td></td>
<td>fadd.d f17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f8</td>
<td>bne</td>
<td></td>
<td>fadd.d f18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td></td>
<td></td>
<td>fadd.d f19</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Loop:**

1. `fld f1, 0(x1)`
2. `fld f2, 8(x1)`
3. `fld f3, 16(x1)`
4. `fld f4, 24(x1)`
5. `add x1, 32`
6. `fadd.d f5, f0, f1`
7. `fadd.d f6, f0, f2`
8. `fadd.d f7, f0, f3`
9. `fadd.d f8, f0, f4`
10. `fsd f5, 0(x2)`
11. `fsd f6, 8(x2)`
12. `fsd f7, 16(x2)`
13. `add x2, 32`
14. `fsd f8, -8(x2)`
15. `bne x1, x3, loop`
## Software Pipelining

### Unroll 4 ways first

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>fld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fld f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd.d f5</td>
<td>fld f1</td>
<td></td>
<td></td>
<td>fadd.d f5</td>
<td></td>
</tr>
<tr>
<td>fadd.d f6</td>
<td>fld f2</td>
<td></td>
<td></td>
<td>fadd.d f6</td>
<td></td>
</tr>
<tr>
<td>fadd.d f7</td>
<td>fld f3</td>
<td></td>
<td></td>
<td>fadd.d f7</td>
<td></td>
</tr>
<tr>
<td>fadd.d f8</td>
<td>fld f4</td>
<td></td>
<td></td>
<td>fadd.d f8</td>
<td></td>
</tr>
<tr>
<td>fsd f5</td>
<td>fadd.d f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd f6</td>
<td>fadd.d f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd f7</td>
<td>fadd.d f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fsd f8</td>
<td>fadd.d f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**prolog**
- loop: fld f1, 0(x1)
- fld f2, 8(x1)
- fld f3, 16(x1)
- fld f4, 24(x1)
- add x1, 32
- fadd.d f5, f0, f1
- fadd.d f6, f0, f2
- fadd.d f7, f0, f3
- fadd.d f8, f0, f4
- fsd f5, 0(x2)
- fsd f6, 8(x2)
- fsd f7, 16(x2)
- add x2, 32
- fsd f8, -8(x2)
- bne x1, x3, loop

**iterate**
- add x1
- add x2
- bne x1, x3, loop

**epilog**
- bne
**Software Pipelining**

**Unroll 4 ways first**

```
loop: 
  fld f1, 0(x1)
  fld f2, 8(x1)
  fld f3, 16(x1)
  fld f4, 24(x1)
  add x1, 32
  fadd.d f5, f0, f1
  fadd.d f6, f0, f2
  fadd.d f7, f0, f3
  fadd.d f8, f0, f4
  fsd f5, 0(x2)
  fsd f6, 8(x2)
  fsd f7, 16(x2)
  add x2, 32
  fsd f8, -8(x2)
  bne x1, x3, 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>fld f1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fld f2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fld f3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add x1</td>
<td></td>
<td>fld f4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fld f1</td>
<td></td>
<td>fadd.d f5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fld f2</td>
<td></td>
<td>fadd.d f6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fld f3</td>
<td></td>
<td>fadd.d f7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add x1</td>
<td></td>
<td>fld f4</td>
<td></td>
<td>fadd.d f8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add x1</td>
<td>bne</td>
<td>fld f4</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>
```

**How many FLOPS/cycle?**

---

November 6, 2023

MIT 6.5900 Fall 2023
Software Pipelining

Unroll 4 ways first

How many FLOPS/cycle?

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

![Diagram showing three groups: Group 1, Group 2, Group 3]
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
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

Four basic blocks

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
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
Inst 2
br a==b, b2
Inst 3
Inst 4
br b3
Inst 5
Inst 6
Inst 7
Inst 8

b0:
if

b1:
else

b2:
then

b3:

Four basic blocks

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
```
IA-64 Speculative Execution

Problem: Branches restrict compiler code motion

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
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
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

Load r1
Use r1
Inst 3

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

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

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

Load r1
Use r1
Inst 3

Inst 1
Inst 2
br a==b, b2

Inst 1
Inst 2
br a==b, b2

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

Inst 1
Inst 2
br a==b, b2

Load r1
Use r1
Inst 3
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

Particularly useful for scheduling long latency loads early
IA-64 Data Speculation

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

L17-34
IA-64 Data Speculation

Problem: Possible memory hazards limit code scheduling

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

Inst 1
Inst 2
Store (addr1)

Load r1, (addr2)
Use r1
Inst 3

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

Data speculative load adds address to address check table

Load.a r1, (addr2)
Inst 1
Inst 2
Store (addr1)

Load.c (addr2)
Use r1
Inst 3

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 VLIW-inspired techniques to traditional RISC/CISC processor architectures?
Thank you!

Next Lecture: Vector Processors

Source: https://en.m.wikipedia.org/wiki/File:Itanium_Sales_Forecasts_edit.png