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.

Matrix A
Matrix B

Decoder

Control lines to ALU, MUXs, Registers
Microcoded Microarchitecture

- Memory (RAM)
  - Data
  - Addr
- Datapath
- μcontroller (ROM)
  - busy?
  - zero?
  - opcode
- enMem
- MemWrt

April 29, 2021
MIT 6.823 Spring 2021
Microcoded Microarchitecture

- **μcontroller (ROM)**: Holds fixed microcode instructions.
- **Datapath**:
  - **Data**
  - **Addr**
- **Memory (RAM)**: Involves signals such as **busy?**, **zero?**, **opcode**, **enMem**, and **MemWrt**.
Microcoded Microarchitecture

- \( \mu \text{controller (ROM)} \)
- Datapath
- Memory (RAM)

- busy?
- zero?
- opcode

- Holds fixed microcode instructions

- Holds user program written in macrocode instructions (e.g., MIPS, x86, etc.)

- Data
- Addr

- enMem
- MemWrt
A Bus-based Datapath for MIPS
A Bus-based Datapath for MIPS
A Bus-based Datapath for MIPS

Diagram with labels:
- Opcode: ldIR
- OpSel: ldA, ldB
- ALU control
- ALU
- RegSel
- ExtSel
- enImm
- enALU
- RegWrt
- enReg
- Bus: 32
- 32-bit Reg
- 32 GPRs + PC ...
- Addr
- data
- 31 (Link)
- 32 (PC)
- rd
- rt
- rs
- zero?
A Bus-based Datapath for MIPS
A Bus-based Datapath for MIPS

Microinstruction: register to register transfer (17 control signals)

MA $\leftarrow$ PC means RegSel = PC; enReg=yes; IdMA= yes

B $\leftarrow$ Reg[rt] 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
Memory Module

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

- Opcode
- ext
- absolute
- op-group
- \( \mu PC \)
- \( \mu PC + 1 \)
- jumps
- \( \mu PCSrc \)
- absolute
- busy
- zero
- \( \mu PC \)
- \( \mu PC + 1 \)
- address
- data
- Control ROM
- Control Signals (17)
Microcode Controller

Input encoding reduces ROM height

Opcode

ext

control signals (17)

 Opcode

Absolute

Jump logic

Busy

Zero

OPC (state)

Address

Data

Control ROM

Control Signals (17)
Microcode Controller

Control Signals (17)

 Opcode

 ext

 absolute

 op-group

 \( \mu PC \) \( \mu PC+1 \)

 jump

 logic

 zero

 busy

 \( \mu PC_{Src} \)

 input encoding reduces ROM height

 address

 Control ROM

 data

 next-state encoding reduces ROM width

 Control Signals (17)
Microcode Controller

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

Control Signals (17)

Control ROM

address

data

jump logic

input encoding reduces ROM height

next-state encoding reduces ROM width

 Opcode \rightarrow ext

absolute

op-group

\( \mu \text{PC} \)

\( \mu \text{PC} +1 \)

+1

\( \mu \text{PCSrc} \)

zero

busy
**Jump Logic**

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

- next \Rightarrow \mu\text{PC}+1
- spin \Rightarrow \text{if (busy) then } \mu\text{PC} \text{ else } \mu\text{PC}+1
- fetch \Rightarrow \text{absolute}
- dispatch \Rightarrow \text{op-group}
- feqz \Rightarrow \text{if (zero) then } \text{absolute} \text{ else } \mu\text{PC}+1
- fnez \Rightarrow \text{if (zero) then } \mu\text{PC}+1 \text{ else } \text{absolute}
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></td>
<td></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

State  Control points  next-state

fetch₀  MA ← PC  next
fetch₁  IR ← Memory
fetch₂  A ← PC
fetch₃  PC ← A + 4

...  

ALU₀  A ← Reg[rs]
ALU₁  B ← Reg[rt]
ALU₂  Reg[rd] ← func(A,B)

ALUᵢ₀  A ← Reg[rs]
ALUᵢ₁  B ← sExt_{16}(Imm)
ALUᵢ₂  Reg[rd] ← Op(A,B)
Instruction Fetch

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

...  

ALU\_0  
A $\leftarrow$ Reg[rs]  

ALU\_1  
B $\leftarrow$ Reg[rt]  

ALU\_2  
Reg[rd] $\leftarrow$ func(A,B)  

ALU\_i\_0  
A $\leftarrow$ Reg[rs]  

ALU\_i\_1  
B $\leftarrow$ sExt\_16(Imm)  

ALU\_i\_2  
Reg[rd] $\leftarrow$ Op(A,B)

---

Diagram showing the control flow and ALU operations.
Instruction Fetch

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

\[\ldots\]

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

ALU\(_i_0\) | A \(\leftarrow\) Reg[rs] | next
ALU\(_i_1\) | B \(\leftarrow\) sExt\(_{16}\)(Imm) | spin
ALU\(_i_2\) | Reg[rd] \(\leftarrow\) Op(A,B) | dispatch

[Diagrams and illustrations related to 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>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   | A ← Reg[rs]   
ALU₁   | B ← Reg[rt]   
ALU₂   | Reg[rd] ← func(A,B) 

ALUi₀   | A ← Reg[rs] 
ALUi₁  | B ← sExt₁₆(Imm) 
ALUi₂  | Reg[rd] ← Op(A,B) 

![Diagram](image.png)
# 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    | A ← Reg[rs]                  | next       |
ALU₁    | B ← Reg[rt]                  | next       |
ALU₂    | Reg[rd]←func(A,B)            |            |

ALUi₀   | A ← Reg[rs]                  | next       |
ALUi₁   | B ← sExt₁₆(Imm)              | next       |
ALUi₂   | Reg[rd]← Op(A,B)             |            |

![Diagram of instruction fetch process](image)
Instruction Fetch

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

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

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&lt;sub&gt;0&lt;/sub&gt;</td>
<td>MA $\leftarrow$ PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch&lt;sub&gt;1&lt;/sub&gt;</td>
<td>IR $\leftarrow$ Memory</td>
<td>spin</td>
</tr>
<tr>
<td>fetch&lt;sub&gt;2&lt;/sub&gt;</td>
<td>A $\leftarrow$ PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch&lt;sub&gt;3&lt;/sub&gt;</td>
<td>PC $\leftarrow$ A + 4</td>
<td>dispatch</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU&lt;sub&gt;0&lt;/sub&gt;</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALU&lt;sub&gt;1&lt;/sub&gt;</td>
<td>B $\leftarrow$ Reg[rt]</td>
<td>next</td>
</tr>
<tr>
<td>ALU&lt;sub&gt;2&lt;/sub&gt;</td>
<td>Reg[rd] $\leftarrow$ func(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>ALU&lt;sub&gt;i0&lt;/sub&gt;</td>
<td>A $\leftarrow$ Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALU&lt;sub&gt;i1&lt;/sub&gt;</td>
<td>B $\leftarrow$ sExt&lt;sub&gt;16&lt;/sub&gt;(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>ALU&lt;sub&gt;i2&lt;/sub&gt;</td>
<td>Reg[rd] $\leftarrow$ Op(A,B)</td>
<td>next</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>LW(_0)</td>
<td>A (\leftarrow) Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>LW(_1)</td>
<td>B (\leftarrow) sExt(_{16})(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>LW(_2)</td>
<td>MA (\leftarrow) A+B</td>
<td>next</td>
</tr>
<tr>
<td>LW(_3)</td>
<td>Reg[rt] (\leftarrow) Memory</td>
<td>spin</td>
</tr>
<tr>
<td>LW(_4)</td>
<td></td>
<td>fetch</td>
</tr>
<tr>
<td>SW(_0)</td>
<td>A (\leftarrow) Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>SW(_1)</td>
<td>B (\leftarrow) sExt(_{16})(Imm)</td>
<td>next</td>
</tr>
<tr>
<td>SW(_2)</td>
<td>MA (\leftarrow) A+B</td>
<td>next</td>
</tr>
<tr>
<td>SW(_3)</td>
<td>Memory (\leftarrow) Reg[rt]</td>
<td>spin</td>
</tr>
<tr>
<td>SW(_4)</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><code>BEQZ_0</code></td>
<td><code>A ← Reg[rs]</code></td>
<td><code>next</code></td>
</tr>
<tr>
<td><code>BEQZ_1</code></td>
<td></td>
<td><code>fnez</code></td>
</tr>
<tr>
<td><code>BEQZ_2</code></td>
<td><code>A ← PC</code></td>
<td><code>next</code></td>
</tr>
<tr>
<td><code>BEQZ_3</code></td>
<td><code>B ← sExt_{16}(Imm&lt;&lt;2)</code></td>
<td><code>next</code></td>
</tr>
<tr>
<td><code>BEQZ_4</code></td>
<td><code>PC ← A+B</code></td>
<td><code>fetch</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BNEZ_0</code></td>
<td><code>A ← Reg[rs]</code></td>
<td><code>next</code></td>
</tr>
<tr>
<td><code>BNEZ_1</code></td>
<td></td>
<td><code>feqz</code></td>
</tr>
<tr>
<td><code>BNEZ_2</code></td>
<td><code>A ← PC</code></td>
<td><code>next</code></td>
</tr>
<tr>
<td><code>BNEZ_3</code></td>
<td><code>B ← sExt_{16}(Imm&lt;&lt;2)</code></td>
<td><code>next</code></td>
</tr>
<tr>
<td><code>BNEZ_4</code></td>
<td><code>PC ← A+B</code></td>
<td><code>fetch</code></td>
</tr>
</tbody>
</table>
# Jumps

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_0$</td>
<td>$A \leftarrow PC$</td>
<td>next</td>
</tr>
<tr>
<td>$J_1$</td>
<td>$B \leftarrow IR$</td>
<td>next</td>
</tr>
<tr>
<td>$J_2$</td>
<td>$PC \leftarrow \text{JumpTarg}(A,B)$</td>
<td>fetch</td>
</tr>
<tr>
<td>$JR_0$</td>
<td>$A \leftarrow \text{Reg[rs]}$</td>
<td>next</td>
</tr>
<tr>
<td>$JR_1$</td>
<td>$PC \leftarrow A$</td>
<td>fetch</td>
</tr>
<tr>
<td>$JAL_0$</td>
<td>$A \leftarrow PC$</td>
<td>next</td>
</tr>
<tr>
<td>$JAL_1$</td>
<td>$\text{Reg[31]} \leftarrow A$</td>
<td>next</td>
</tr>
<tr>
<td>$JAL_2$</td>
<td>$B \leftarrow IR$</td>
<td>next</td>
</tr>
<tr>
<td>$JAL_3$</td>
<td>$PC \leftarrow \text{JumpTarg}(A,B)$</td>
<td>fetch</td>
</tr>
<tr>
<td>$JALR_0$</td>
<td>$A \leftarrow PC$</td>
<td>next</td>
</tr>
<tr>
<td>$JALR_1$</td>
<td>$B \leftarrow \text{Reg[rs]}$</td>
<td>next</td>
</tr>
<tr>
<td>$JALR_2$</td>
<td>$\text{Reg[31]} \leftarrow A$</td>
<td>next</td>
</tr>
<tr>
<td>$JALR_3$</td>
<td>$PC \leftarrow B$</td>
<td>fetch</td>
</tr>
</tbody>
</table>
Very Long Instruction Word (VLIW) Processors
Sequential source code

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

Superscalar compiler

*Find independent operations*
Sequential ISA Bottleneck

Sequential source code

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

Superscalar compiler

Find independent operations

Schedule operations
Sequential ISA Bottleneck

Sequential source code

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

Sequential source code

Superscalar compiler

Find independent operations

Schedule operations

Sequential machine code

Superscalar processor

Check instruction dependencies

a = foo(b);
for (i=0, i<
Sequential ISA Bottleneck

Sequential source code

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

Superscalar compiler

Find independent operations

Schedule operations

Sequential machine code

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
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
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 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></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>
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
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
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>Add</th>
<th>Int 1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<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>
<td></td>
</tr>
<tr>
<td>SD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Loop:  add r1  ld  
       fadd  
       sd  
       bne r1, r3, loop
## Loop Execution

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

### Compile

1. **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>Int2</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>Add r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Id</td>
<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>
<td></td>
</tr>
<tr>
<td>Add r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The loop is executed with the given instructions and schedule.
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 Execution

Int1  Int 2  M1  M2  FP+  FPx
loop:  add r1  ld  
       fadd  
       add r2  bne  sd  

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

How many FP ops/cycle?
### Loop Execution

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

---

#### Compile

```plaintext
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>addr</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>

---

#### How many FP ops/cycle?

1 fadd / 8 cycles = 0.125
Loop Unrolling

Unroll inner loop to perform 4 iterations at once

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

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

Unroll inner loop to perform 4 iterations at once

Is this code 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 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

<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>
</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
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
```

```
loop:
   ld f1
   ld f2
   ld f3
   ld f4
   ```
### 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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f5</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>
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
---  ---  ---  ---  ---  ---
loop:  ld f1
       ld f2
       ld f3
add r1, 32
       ld f4
       fadd f5

L19-22
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></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>ld f1</td>
<td></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>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f4</td>
<td>add r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ld f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fadd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>fadd f6</td>
<td></td>
<td></td>
<td></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>
</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>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>add r1</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>
<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 f7</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>
</tr>
</tbody>
</table>
Scheduling Loop Unrolled Code

Unroll 4 ways

Unroll the loop to improve performance by spreading instructions across multiple iterations. This can help reduce the number of stalls and improve the use of available resources.

```
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

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

## Schedule

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

How many FLOPS/cycle?
## Scheduling Loop Unrolled Code

### Unroll 4 ways

Loop:  
\[
\begin{align*}
\text{ld } f1, & \ 0(r1) \\
\text{ld } f2, & \ 8(r1) \\
\text{ld } f3, & \ 16(r1) \\
\text{ld } f4, & \ 24(r1) \\
\text{add } r1, & \ 32 \\
fadd & \ f5, f0, f1 \\
fadd & \ f6, f0, f2 \\
fadd & \ f7, f0, f3 \\
fadd & \ f8, f0, f4 \\
\text{sd } f5, & \ 0(r2) \\
\text{sd } f6, & \ 8(r2) \\
\text{sd } f7, & \ 16(r2) \\
\text{sd } f8, & \ 24(r2) \\
\text{add } r2, & \ 32 \\
bne & \ r1, r3, \text{loop}
\end{align*}
\]

### 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>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>fadd f5</td>
<td></td>
<td></td>
<td></td>
<td>fadd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f6</td>
<td></td>
<td></td>
<td></td>
<td>fadd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f7</td>
<td></td>
<td></td>
<td></td>
<td>fadd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f8</td>
<td></td>
<td></td>
<td></td>
<td>fadd</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>add r2</td>
<td></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>
<td></td>
</tr>
<tr>
<td>sd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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>
</tbody>
</table>

---

Int1 Int 2 M1 M2 FP+ FPx

L19-23
## 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></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>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>add r1</td>
<td>ld f4</td>
<td></td>
<td></td>
<td>fadd f5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fadd f6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fadd f7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fadd f8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sd f5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sd f6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>add r2</td>
<td>sd f7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bne</td>
<td>sd f8</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

**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`

### Pipeline Stages

**Int1**
- `ld f1`
- `ld f2`
- `ld f3`
- `ld f4`
- `add r1`
- `ld f4`
- `ld f1`
- `ld f2`
- `ld f3`
- `add r1`
- `ld f4`
- `add r2`
- `bne`  

**Int 2**
- `ld f2`
- `ld f3`
- `ld f4`
- `fadd f5`
- `fadd f6`
- `fadd f7`
- `fadd f8`
- `sd f5`
- `sd f6`
- `sd f7`
- `sd f8`
- `bne`

**M1**

**M2**

**FP+**

**FPx**

---

April 29, 2021  
MIT 6.823 Spring 2021  
L19-23
## 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></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>ld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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>add r1</td>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f1</td>
<td></td>
<td></td>
<td></td>
<td>fadd f5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f2</td>
<td></td>
<td></td>
<td></td>
<td>fadd f6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f3</td>
<td></td>
<td></td>
<td></td>
<td>fadd f7</td>
<td></td>
</tr>
<tr>
<td>add r1</td>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
<td>fadd f8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f1</td>
<td></td>
<td></td>
<td></td>
<td>sd f5</td>
<td>fadd f5</td>
</tr>
<tr>
<td></td>
<td>ld f2</td>
<td></td>
<td></td>
<td></td>
<td>sd f6</td>
<td>fadd f6</td>
</tr>
<tr>
<td>add r2</td>
<td>ld f3</td>
<td></td>
<td></td>
<td></td>
<td>sd f7</td>
<td>fadd f7</td>
</tr>
<tr>
<td>add r1</td>
<td>bne</td>
<td>ld f4</td>
<td></td>
<td></td>
<td>sd f8</td>
<td>fadd f8</td>
</tr>
<tr>
<td>add r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sd f7</td>
<td>fadd f7</td>
</tr>
<tr>
<td>bne</td>
<td></td>
<td></td>
<td></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>
<td>sd f5</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:**

<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>Prolog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterate</td>
<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>
<td></td>
</tr>
</tbody>
</table>

- `ld f1`
- `ld f2`
- `ld f3`
- `ld f4`
- `fadd f5`
- `fadd f6`
- `fadd f7`
- `fadd f8`
- `sd f5`
- `sd f6`
- `sd f7`
- `sd f8`
- `add r1`
- `add r2`
- `bne`

**Int1**
- `Int 2`
- `M1`
- `M2`
- `FP+`
- `FPx`

**Notes:**
- Unroll 4 ways first
- Prolog
- Iterate
- Loop
- Epilog

---

*April 29, 2021*
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

How many FLOPS/cycle?
### 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>prolog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f1</td>
<td>ld f2</td>
<td>ld f3</td>
<td>ld f4</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 f1</td>
<td>fadd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f2</td>
<td>fadd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f3</td>
<td>fadd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1</td>
<td>ld f4</td>
<td>fadd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>iterate</td>
<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>
<td></td>
</tr>
<tr>
<td>ld f1</td>
<td>sd f5</td>
<td>fadd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f2</td>
<td>sd f6</td>
<td>fadd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2</td>
<td>ld f3</td>
<td>sd f7</td>
<td>fadd f7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1</td>
<td>bne</td>
<td>ld f4</td>
<td>sd f8</td>
<td>fadd 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>
<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>epilog</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>

### 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
• 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
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

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

```
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
```

```
Inst 1     ||   (p1) Inst 3     ||   (p2) Inst 5
(p1) Inst 4     ||   (p2) Inst 6
Inst 7
Inst 8
```

Mahlke et al, ISCA95: On average >50% branches removed
Fully Bypassed Datapath

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
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
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
  - Chk.s r1
  - Use r1
  - Inst 3

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

Particularly useful for scheduling long latency loads early
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

Inst 1
Inst 2
Store

Load r1
Use r1
Inst 3

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

Inst 1
Inst 2
Store

Load r1
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
Inst 1
Inst 2
Store

Load.c
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 the VLIW-inspired techniques to traditional RISC/CISC processor architectures?
Thank you!

Next Lecture: Vector Processors