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 8, 2021
Microcontrol Unit [Maurice Wilkes, 1954]

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

<table>
<thead>
<tr>
<th>op code</th>
<th>conditional flip-flop</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td></td>
</tr>
</tbody>
</table>

Matrix A

Matrix B

Control lines to ALU, MUXs, Registers

Decoder

Next state
Microcoded Microarchitecture

Microcontroller (ROM)

Datapath

Memory (RAM)

Data

Addr

enMem

MemWrt

busy?

zero?

opcode
Microcaded Microarchitecture

- **μcontroller (ROM)**
  - Inputs: `busy?`, `zero?`, `opcode`
  - Outputs: fixed microcode instructions

- **Datapath**
  - Inputs: `Data`, `Addr`

- **Memory (RAM)**
  - Inputs: `enMem`, `MemWrt`
Microcoded Microarchitecture

- **μcontroller (ROM)**
  - Holds fixed microcode instructions
  - Inputs: busy?, zero?, opcode

- **Datapath**
  - Inputs: Data, Addr

- **Memory (RAM)**
  - Inputs: enMem, MemWrt
  - Outputs: busy?, zero?, opcode

- **Memory**
  - Holds user program written in macrocode instructions (e.g., MIPS, x86, etc.)
A Bus-based Datapath for MIPS
A Bus-based Datapath for MIPS

[Diagram of a bus-based datapath for MIPS, showing components like Opcode, OpSel, ALU, and Bus 32.]
A Bus-based Datapath for MIPS

![Diagram of a bus-based datapath for MIPS](image)
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 \text{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
Assumption: Memory operates asynchronously and is slow compared to Reg-to-Reg transfers
Microcode Controller

![Diagram of microcode controller with control signals and ROM integration](image-url)
Microcode Controller

Opcode \rightarrow ext

absolute

op-group

\mu PC (state)

\mu PC \rightarrow \mu PC +1

+1

\mu PCSrc

jump logic

address

data

input encoding reduces ROM height

Control Signals (17)
Microcode Controller

Opcode → ext

op-group

μPC (state)

μPC
μPC+1

+1

μPCSr

c control logic

zero

busy

address

Control ROM

data

Control Signals (17)

input encoding reduces ROM height

next-state encoding reduces ROM width

jump logic

[Diagram showing control signals and logic for microcode controller]
Microcode Controller

$\mu\text{JumpType} = \text{next} \mid \text{spin} \mid \text{fetch} \mid \text{dispatch} \mid \text{feqz} \mid \text{fnez}$

Control ROM

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 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 ← PC</td>
<td></td>
</tr>
<tr>
<td>fetch_1</td>
<td>IR ← Memory</td>
<td></td>
</tr>
<tr>
<td>fetch_2</td>
<td>A ← PC</td>
<td></td>
</tr>
<tr>
<td>fetch_3</td>
<td>PC ← A + 4</td>
<td></td>
</tr>
</tbody>
</table>

...  

### 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></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>
</tbody>
</table>

...  

$ALU_0$  
A $\leftarrow \text{Reg[rs]}$  

$ALU_1$  
B $\leftarrow \text{Reg[rt]}$  

$ALU_2$  
Reg[rd] $\leftarrow \text{func}(A,B)$  

$ALU_i_0$  
A $\leftarrow \text{Reg[rs]}$  

$ALU_i_1$  
B $\leftarrow \text{sExt}_{16}(\text{Imm})$  

$ALU_i_2$  
Reg[rd] $\leftarrow \text{Op}(A,B)$
# 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 spin</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>
</tbody>
</table>

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

ALU\(_{i0}\) A $\leftarrow$ Reg[rs]  
ALU\(_{i1}\) B $\leftarrow$ sExt\(_{16}\)(Imm)  
ALU\(_{i2}\) 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(_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) |  

![Instruction Fetch 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>
</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)
# 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[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALU(_1)</td>
<td>B ← Reg[rt]</td>
<td></td>
</tr>
<tr>
<td>ALU(_2)</td>
<td>Reg[rd] ← func(A,B)</td>
<td></td>
</tr>
<tr>
<td>ALU(_i_0)</td>
<td>A ← Reg[rs]</td>
<td></td>
</tr>
<tr>
<td>ALU(_i_1)</td>
<td>B ← sExt(_{16})(Imm)</td>
<td></td>
</tr>
<tr>
<td>ALU(_i_2)</td>
<td>Reg[rd] ← Op(A,B)</td>
<td></td>
</tr>
</tbody>
</table>

## Diagram

The diagram illustrates the control flow and data flow in the instruction fetch phase. The pipeline stages include IR (Instruction Register), ALU (Arithmetic Logic Unit), and Memory. Each stage has inputs and outputs, with specific signals such as Opcode, IdIR, OpSel, IdA, IdB, zero?, rd, rs, Imm Ext, enImm, enALU, data, Addr, RegSel, MemWrt, and enMem. The pipeline stages are connected by signals that indicate the execution of instructions, memory access, and data movement.
**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)         |            |
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 | next
fetch\(_3\) | PC ← A + 4 | dispatch
...

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

ALU\(_i\)_0 | A ← Reg[rs] |
ALU\(_i\)_1 | B ← sExt\(_{16}\)(Imm) |
ALU\(_i\)_2 | Reg[rd] ← Op(A,B) |
## Instruction Fetch

### State Control points next-state

<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) | fetch     |

| ALUᵢ₀      | A ← Reg[rs]    | next       |
| ALUᵢ₁      | B ← sExt₁₆(Imm) | next       |
| ALUᵢ₂      | Reg[rd] ← Op(A,B) | fetch     |
# Load & Store

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

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQZ₀</td>
<td>$A \leftarrow \text{Reg}[rs]$</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ₁</td>
<td>$A \leftarrow \text{PC}$</td>
<td>fnez</td>
</tr>
<tr>
<td>BEQZ₂</td>
<td>$A \leftarrow \text{PC}$</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ₃</td>
<td>$B \leftarrow \text{sExt}_{16}(\text{Imm}&lt;&lt;2)$</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ₄</td>
<td>$\text{PC} \leftarrow A+B$</td>
<td>fetch</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNEZ₀</td>
<td>$A \leftarrow \text{Reg}[rs]$</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ₁</td>
<td>$A \leftarrow \text{PC}$</td>
<td>feqz</td>
</tr>
<tr>
<td>BNEZ₂</td>
<td>$A \leftarrow \text{PC}$</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ₃</td>
<td>$B \leftarrow \text{sExt}_{16}(\text{Imm}&lt;&lt;2)$</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ₄</td>
<td>$\text{PC} \leftarrow A+B$</td>
<td>fetch</td>
</tr>
</tbody>
</table>
## Jumps

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>J₀</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>J₁</td>
<td>B ← IR</td>
<td>next</td>
</tr>
<tr>
<td>J₂</td>
<td>PC ← JumpTarg(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>JR₀</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>JR₁</td>
<td>PC ← A</td>
<td>fetch</td>
</tr>
<tr>
<td>JAL₀</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>JAL₁</td>
<td>Reg[31] ← 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 ← PC</td>
<td>next</td>
</tr>
<tr>
<td>JALR₁</td>
<td>B ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>JALR₂</td>
<td>Reg[31] ← A</td>
<td>next</td>
</tr>
<tr>
<td>JALR₃</td>
<td>PC ← 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

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

Superscalar compiler

Find independent operations
Sequential ISA Bottleneck

Sequential source code

\[ a = \text{foo}(b); \]
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
Sequential ISA Bottleneck

Sequential source code

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

Superscalar compiler

Find independent operations
Schedule operations

Superscalar processor

Check instruction dependencies

Sequential machine code
Sequential ISA Bottleneck

Sequential source code

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

Sequential machine code

Superscalar compiler

Find independent operations

Schedule operations

Superscalar processor

Check instruction dependencies

Schedule execution

L17-16
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

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
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++)

**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++)
    \[ B[i] = A[i] + C; \]

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>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 r1</td>
<td>ld</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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++)

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

Compile

Schedule

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

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

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

Compile

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
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++)
    \[ B[i] = A[i] + C; \]

Compile

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

Schedule

How many FP ops/cycle?
Loop Execution

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

Compile

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

Schedule

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

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

Unroll inner loop to perform 4 iterations at once
Loop Unrolling

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

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

Unroll inner loop to perform 4 iterations at once

Is this code correct?
Loop Unrolling

Unroll inner loop to perform 4 iterations at once

Is this code correct?

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

Unroll 4 ways

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

```
<table>
<thead>
<tr>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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>ld f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</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>
<tr>
<td>add r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne r1, r3, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Scheduling Loop Unrolled Code

Unroll 4 ways

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

Schedule

<table>
<thead>
<tr>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>loop:</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>add r1</td>
<td>ld f4</td>
<td>fadd f5</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>ld f4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1</td>
<td></td>
<td></td>
<td>fadd f5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fadd f6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fadd f7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>fadd f8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2, 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne r1, r3, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scheduling Loop Unrolled Code

Unroll 4 ways

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

loop:

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

L17-22
### 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>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd r1</td>
<td>ld f4</td>
<td></td>
<td>fadd f5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f6</td>
<td>fadd f6</td>
<td></td>
<td>fadd f7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f7</td>
<td>fadd f8</td>
<td></td>
<td>fadd f8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f5</td>
<td>sd f5</td>
<td></td>
<td>sd f6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f6</td>
<td>sd f6</td>
<td></td>
<td>sd f7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f7</td>
<td>sd f7</td>
<td></td>
<td>sd f8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2</td>
<td>bne</td>
<td></td>
<td>sd f8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

L17-22
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**

How many FLOPS/cycle?
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?

4 fadds / 11 cycles = 0.36
### 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
```
**Software Pipelining**

*Unroll 4 ways first*

<table>
<thead>
<tr>
<th></th>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f1</td>
<td></td>
<td>ld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f2</td>
<td></td>
<td>ld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f3</td>
<td></td>
<td>ld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f4</td>
<td></td>
<td>ld f4</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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>
</tbody>
</table>

```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
```

November 8, 2021

MIT 6.823 Fall 2021
Software Pipelining

**Unroll 4 ways first**

<table>
<thead>
<tr>
<th></th>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f1, 0(r1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f2, 8(r1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f3, 16(r1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ld f4, 24(r1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1, 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f5, f0, f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f6, f0, f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f7, f0, f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fadd f8, f0, f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f5, 0(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f6, 8(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f7, 16(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2, 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sd f8, -8(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne r1, r3, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

L17-23
## 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></td>
<td></td>
<td></td>
<td><code>ld f1</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>ld f2</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>ld f3</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>add r1</code></td>
<td><code>ld f4</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>ld f1</code></td>
<td><code>fadd f5</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>ld f2</code></td>
<td><code>fadd f6</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>ld f3</code></td>
<td><code>fadd f7</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>add r1</code></td>
<td><code>ld f4</code></td>
<td><code>fadd f8</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>ld f1</code></td>
<td><code>sd f5</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>ld f2</code></td>
<td><code>sd f6</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>add r2</code></td>
<td><code>ld f3</code></td>
<td><code>sd f7</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>add r1</code></td>
<td><code>bne</code></td>
<td><code>ld f4</code></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><code>add r2</code></td>
<td></td>
<td><code>sd f7</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><code>bne</code></td>
<td></td>
<td><code>sd f8</code></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

Int1  Int 2  M1  M2  FP+  FPx

prolog

iterate

loop:

epilog

Unroll 4 ways first

ld f1  ld f2  ld f3  ld f4  fadd f5  fadd f6  fadd f7  fadd f8
add r1  ld f4  ld f4  ld f4

ld f1  ld f2  ld f3  ld f4  fadd f5  fadd f6  fadd f7  fadd f8
add r1  ld f4  ld f4  ld f4

ld f1  ld f2  ld f3  ld f4  fadd f5  fadd f6  fadd f7  fadd f8
add r1  ld f4  ld f4  ld f4

add r2  ld f3  sd f7  fadd f7
add r1  bne  ld f4  sd f8  fadd f8
add r2  sd f7  fadd f7
bne  sd f8  fadd f8
bne  sd f5
d

November 8, 2021  MIT 6.823 Fall 2021  L17-23
Software Pipelining

Unroll 4 ways first

How many FLOPS/cycle?
**Software Pipelining**

**Unroll 4 ways first**

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

**How many FLOPS/cycle?**

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

**Loop Unrolling**

- **Startup overhead**
- **Wind-down overhead**

**Software Pipelining**

**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 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 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

- 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
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
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

\[
\begin{array}{ll}
b0: & \text{Inst 1} \quad \text{if} \\
     & \text{Inst 2} \\
     & \text{br a==b, b2} \\
\hline
b1: & \text{Inst 3} \quad \text{else} \\
     & \text{Inst 4} \\
     & \text{br b3} \\
\hline
b2: & \text{Inst 5} \quad \text{then} \\
     & \text{Inst 6} \\
\hline
b3: & \text{Inst 7} \\
     & \text{Inst 8} \\
\end{array}
\]

Predication

\[
\begin{array}{ll}
\text{Inst 1} \\
\text{Inst 2} \\
p1,p2 \leftarrow \text{cmp(a==b)} \\
(p1) \text{Inst 3} || (p2) \text{Inst 5} \\
(p1) \text{Inst 4} || (p2) \text{Inst 6} \\
\text{Inst 7} \\
\text{Inst 8} \\
\end{array}
\]

One basic block

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

Where does predication fit in?

PC for JAL, ...

ASrc

BSrc

Stall

PC

0x4 Add

Addr Inst Memory

IR

Addr Inst

D D D D

Imm Ext

rd1 rs1 rs2

Ext

ws wd rd2 GPRs

A A A

Ir

E M

ALU

Y

B

MD1 MD2

R

V w e

31

nop

W

IR

Wdata

Memory Data

V w e

addr

rdata

wdata

We
IA-64 Speculative Execution

Problem: Branches restrict compiler code motion
IA-64 Speculative Execution

Problem: Branches restrict compiler code motion

![Diagram of IA-64 execution flow](image)

Can’t move load above branch because might cause spurious exception
IA-64 Speculative Execution

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don’t cause exceptions

Inst 1
Inst 2
br a==b, b2

Load r1
Use r1
Inst 3

Can’t move load above branch because might cause spurious exception
IA-64 Speculative Execution

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don’t cause exceptions

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

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

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

Problem: Branches restrict compiler code motion

Solution: Speculative operations that don’t cause exceptions

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

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

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

Particularly useful for scheduling long latency loads early
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

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

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*