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

• 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
MIT 6.823 Fall 2021
Embed the control logic state table in a read-only memory array

Microcontrol Unit [Maurice Wilkes, 1954]

Decoder

\[\mu\] address

op code
conditional flip-flop

Next state

Control lines to ALU, MUXs, Registers
Microcoded Microarchitecture

- **μcontroller (ROM)**
  - holds fixed microcode instructions

- **Datapath**

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

- **EnMem**
- **MemWrt**
- **Addr**
- **Data**

Signals:
- **busy?**
- **zero?**
- **opcode**
A Bus-based Datapath for MIPS

Microinstruction: register to register transfer (17 control signals)

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

B $\leftarrow$ Reg[rt] means

MA $\leftarrow$ PC; enReg=yes; ldMA= yes
• Assumption: Memory operates asynchronously and is slow compared to Reg-to-Reg transfers
Microcode Controller

μJumpType = next | spin | fetch | dispatch | feqz | fnez

input encoding reduces ROM height

Control Signals (17)

Opcode → ext

op-group

μPC

μPC+1

+1

jump logic

μPCSrc

zero

busy

absolute

Control ROM

address

data

next-state encoding reduces ROM width

Opcode → μPC (state)
<table>
<thead>
<tr>
<th>Jump Type</th>
<th>μJumpTypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>μPC+1</td>
</tr>
<tr>
<td>spin</td>
<td>if (busy) then μPC else μPC+1</td>
</tr>
<tr>
<td>fetch</td>
<td>absolute</td>
</tr>
<tr>
<td>dispatch</td>
<td>op-group</td>
</tr>
<tr>
<td>feqz</td>
<td>if (zero) then absolute else μPC+1</td>
</tr>
<tr>
<td>fnez</td>
<td>if (zero) then μPC+1 else absolute</td>
</tr>
</tbody>
</table>
Instruction Execution

Execution of a MIPS instruction involves

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

State | Control points | Next-state |
--- | --- | --- |
fetch_0 | MA ← PC | |
fetch_1 | IR ← Memory | |
fetch_2 | A ← PC | |
fetch_3 | PC ← A + 4 | |
...

ALU_0 | A ← Reg[rs] | |
ALU_1 | B ← Reg[rt] | |
ALU_2 | Reg[rd] ← \text{func}(A,B) | |

ALU_i_0 | A ← Reg[rs] | |
ALU_i_1 | B ← sExt_{16}(Imm) | |
ALU_i_2 | Reg[rd] ← \text{Op}(A,B) | |
<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>spin</td>
</tr>
<tr>
<td>LW₃</td>
<td>Reg[rt] ← Memory</td>
<td>fetch</td>
</tr>
<tr>
<td>LW₄</td>
<td></td>
<td></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>spin</td>
</tr>
<tr>
<td>SW₃</td>
<td>Memory ← Reg[rt]</td>
<td>fetch</td>
</tr>
<tr>
<td>SW₄</td>
<td></td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>Control points</td>
<td>next-state</td>
</tr>
<tr>
<td>-------</td>
<td>----------------</td>
<td>------------</td>
</tr>
<tr>
<td>BEQZ_0</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ_1</td>
<td>A ← PC</td>
<td>fnez</td>
</tr>
<tr>
<td>BEQZ_2</td>
<td>B ← sExt_{16}(Imm&lt;&lt;2)</td>
<td>next</td>
</tr>
<tr>
<td>BEQZ_3</td>
<td>PC ← A+B</td>
<td>fetch</td>
</tr>
<tr>
<td>BNEZ_0</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ_1</td>
<td>A ← PC</td>
<td>feqz</td>
</tr>
<tr>
<td>BNEZ_2</td>
<td>B ← sExt_{16}(Imm&lt;&lt;2)</td>
<td>next</td>
</tr>
<tr>
<td>BNEZ_3</td>
<td>PC ← A+B</td>
<td>fetch</td>
</tr>
</tbody>
</table>
## Jumps

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_0$</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>
VAX 11-780 Microcode (1978)

MIT 6.823 Fall 2021
Very Long Instruction Word (VLIW) Processors
Sequential ISA Bottleneck

Sequential source code:

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

Sequential machine code:

Superscalar processor:
- Check instruction dependencies
- Schedule execution

Superscalar compiler:
- Find independent operations
- Schedule operations
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:
- 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

How many FP ops/cycle?

<table>
<thead>
<tr>
<th></th>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r2</td>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Schedule

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

Unroll inner loop to perform 4 iterations at once

Is this code correct?

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

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

<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>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>add r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f5</td>
<td></td>
<td>fadd f5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f6</td>
<td></td>
<td>fadd f6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f7</td>
<td></td>
<td>fadd f7</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f8</td>
<td></td>
<td>fadd f8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sd f7</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></td>
<td>bne r3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sd f8</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
```

How many FLOPS/cycle?

#### 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>loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1</td>
<td>ld f1</td>
<td>ld f4</td>
<td>fadd f5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r2</td>
<td>ld f1</td>
<td>ld f4</td>
<td>fadd f8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fadd f8</td>
</tr>
</tbody>
</table>

**Unroll 4 ways first**

- **Prolog**
  - PROLOG: `ld f1`
  - `ld f2`
  - `ld f3`
  - `ld f4`
- **Loop**
  - `add r1`
  - `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`
  - `bne r1, r3, loop`
- **Epilog**
  - `add r2`
  - `sd f7`
  - `fadd f7`
  - `bne`
  - `sd f8`
  - `fadd f8`
  - `sd f5`
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

How do we know which trace to pick?
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)

- 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

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

```
<table>
<thead>
<tr>
<th>Inst 1</th>
<th>if</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst 2</td>
<td></td>
</tr>
<tr>
<td>br a==b, b2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inst 3</th>
<th>else</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst 4</td>
<td></td>
</tr>
<tr>
<td>br b3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inst 5</th>
<th>then</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst 6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Inst 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst 8</td>
</tr>
</tbody>
</table>
```

Predication

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

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

Four basic blocks
Where does predication fit in?
IA-64 Speculative Execution

Problem: Branches restrict compiler code motion
Solution: Speculative operations that don’t cause exceptions

Particularly useful for scheduling long latency loads early

Load r1
Use r1
Inst 3

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

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

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

Chk.s r1
Use r1
Inst 3

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

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

Next Lecture: Vector Processors