Microcoded and VLIW Processors

Joel Emer
Computer Science & Artificial Intelligence Lab
M.I.T.

http://www.csg.csail.mit.edu/6.823
Microcontrol Unit *Maurice Wilkes, 1954*

*Embed the control logic state table in a memory array*

**Diagram:**
- **op** and **conditional code** flip-flop
- **μ address**
- **Decoder**
- **Matrix A**
- **Matrix B**
- **Control lines to ALU, MUXs, Registers**
- **Next state**
Microcoded Microarchitecture

- **μcontroller (ROM)** holds fixed microcode instructions
- **Memory (RAM)** holds user program written in macrocode instructions (e.g., MIPS, x86, etc.)
- Datapath
  - **Data**
  - **Addr**
  - **enMem**
  - **MemWrt**

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

Microinstruction: register to register transfer (17 control signals)

MA ← PC means RegSel = PC; enReg = yes; ldMA = yes
B ← Reg[rt] means RegSel = rt; enReg = yes; ldB = yes
Memory Module

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

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

Control Signals (17)

Next-state encoding reduces ROM width

Input encoding reduces ROM height

Opcode → ext

absolute

op-group

\[ \mu \text{PC} \]

\[ \mu \text{PC} + 1 \]

+1

\[ \mu \text{PCSrc} \]

Jump logic

Control ROM

Address

Data

Busy

Zero

April 26, 2017

http://www.csg.csail.mit.edu/6.823

Sanchez & Emer
Jump Logic

$\mu\text{PCS}_{\text{rc}} = \text{Case} \quad \mu\text{JumpTypes}$

- 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&lt;sub&gt;0&lt;/sub&gt;</td>
<td>MA ← PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch&lt;sub&gt;1&lt;/sub&gt;</td>
<td>IR ← Memory</td>
<td>spin</td>
</tr>
<tr>
<td>fetch&lt;sub&gt;2&lt;/sub&gt;</td>
<td>A ← PC</td>
<td>next</td>
</tr>
<tr>
<td>fetch&lt;sub&gt;3&lt;/sub&gt;</td>
<td>PC ← A + 4</td>
<td>dispatch</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU&lt;sub&gt;0&lt;/sub&gt;</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALU&lt;sub&gt;1&lt;/sub&gt;</td>
<td>B ← Reg[rt]</td>
<td>next</td>
</tr>
<tr>
<td>ALU&lt;sub&gt;2&lt;/sub&gt;</td>
<td>Reg[rd] ← func(A,B)</td>
<td>fetch</td>
</tr>
<tr>
<td>ALU&lt;sub&gt;i0&lt;/sub&gt;</td>
<td>A ← Reg[rs]</td>
<td>next</td>
</tr>
<tr>
<td>ALU&lt;sub&gt;i1&lt;/sub&gt;</td>
<td>B ← 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] ← Op(A,B)</td>
<td>fetch</td>
</tr>
</tbody>
</table>
# Load & Store

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

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

<table>
<thead>
<tr>
<th>State</th>
<th>Control points</th>
<th>next-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_0$</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>$PC \leftarrow \text{JumpTarg}(A,B)$ fetch</td>
<td></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>$PC \leftarrow \text{JumpTarg}(A,B)$ fetch</td>
<td></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>
Sequential ISA Bottleneck

Sequential source code

Superscalar compiler

Find independent operations

Schedule operations

Sequential machine code

Superscalar processor

Check instruction dependencies

Schedule execution

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

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

#### Compile

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

#### Schedule

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

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

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

Schedule

<table>
<thead>
<tr>
<th></th>
<th>Int1</th>
<th>Int 2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td>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>fadd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>sd f5</td>
<td></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>
<td></td>
</tr>
<tr>
<td></td>
<td>sd f7</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></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 / 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><strong>Prolog</strong></td>
<td></td>
<td></td>
<td><strong>ld f1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>ld f2</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>ld f3</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Loop</strong></td>
<td></td>
<td></td>
<td><strong>ld f4</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>fadd f5</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>fadd f6</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>fadd f7</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>add r1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Iterate</strong></td>
<td></td>
<td></td>
<td><strong>ld f4</strong></td>
<td></td>
<td><strong>fadd f8</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>ld f1</strong></td>
<td><strong>fadd f5</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>ld f2</strong></td>
<td><strong>fadd f6</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>ld f3</strong></td>
<td><strong>fadd f7</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>add r2</strong></td>
<td></td>
<td><strong>ld f3</strong></td>
<td><strong>sd f7</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>add r1</strong></td>
<td></td>
<td><strong>bne</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Epilog</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>ld f4</strong></td>
<td><strong>sd f8</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>fadd f8</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>sd f5</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>sd f6</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>bne</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>sd f7</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>fadd f8</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>sd f5</strong></td>
<td></td>
</tr>
</tbody>
</table>

How many FLOPS/cycle?

**4 fadds / 4 cycles = 1**
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

Basic block
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

• 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

Mahlke et al, ISCA95: On average >50% branches removed
## Predicate Software Pipeline Stages

### Single VLIW Instruction

<table>
<thead>
<tr>
<th>(p1) ld r1</th>
<th>(p2) add r3</th>
<th>(p3) st r4</th>
<th>(p1) bloop</th>
</tr>
</thead>
</table>

### Dynamic Execution

- (p1) ld r1
- (p1) ld r1
- (p1) ld r1
- (p1) ld r1
- (p2) add r3
- (p2) add r3
- (p2) add r3
- (p2) add r3
- (p3) st r4
- (p3) st r4
- (p3) st r4
- (p3) st r4
- (p1) bloop
- (p1) bloop
- (p1) bloop
- (p1) bloop
- (p1) bloop
- (p1) bloop
- (p1) bloop
- (p1) bloop

Software pipeline stages turned on by rotating predicate registers ➔ Much denser encoding of loops
Fully Bypassed Datapath

Where does predication fit in?
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

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

Requires associative hardware in address check table

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

Inst 1
Inst 2
Store

Load r1
Use r1
Inst 3

Inst 1
Load a r1
Inst 1
Inst 2
Store

Load c
Use r1
Inst 3

Inst 3
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 the VLIW-inspired techniques to traditional RISC/CISC processor architectures?
Thank you!