Single-Cycle Implementations

- Matches naturally with the sequential and atomic semantics inherent to most ISAs
  - instantiate programmer-visible state one-for-one
  - map instructions to combinational next-state logic

- But, contrived and inefficient
  - all instructions run as slow as the slowest instruction
  - must provide worst-case combinational resource in parallel as required by any instruction

- Not necessarily the simplest way to implement an ISA

Imagine trying to build a single-cycle implementation of a CISC ISA
Single-Cycle Datapath Analysis

Assume
- memory units (read or write): 200 ps
- ALU and adders: 100 ps
- register file (read or write): 50 ps
- other combinational logic: 0 ps

<table>
<thead>
<tr>
<th>steps</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>resources</td>
<td>mem</td>
<td>RF</td>
<td>ALU</td>
<td>mem</td>
<td>RF</td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>I-type</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>50</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>200</td>
<td>50</td>
<td>600</td>
</tr>
<tr>
<td>SW</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>200</td>
<td>50</td>
<td>550</td>
</tr>
<tr>
<td>Branch</td>
<td>200</td>
<td>50</td>
<td>100</td>
<td>200</td>
<td>50</td>
<td>350</td>
</tr>
<tr>
<td>Jump</td>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>200</td>
</tr>
</tbody>
</table>

Worksheet

(Based on figures from P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.)
Multi-cycle Implementation: Ver 1.0

- Why not let each instruction type take only as much time as it needs
- Idea
  - run a 50 ps clock
  - let each instruction type take as many clock cycles as needed
  - programmer-visible state only updates at the end of an instruction’s cycle-sequence
  - an instruction’s effect is still purely combinational from PVS to PVS

A more realistic alternative to the "variable-length" clock design in the textbook

Multi-Cycle Datapath: Ver 1.0
Sequential Control: Ver 1.0

What about the rest of the control signals?

MicroSequencer: Ver 1.0

- ROM as a combinational logic lookup table

** ROM size grows as $O(2^n)$ as the number of inputs

** ROM size grows as $O(m)$ as the number of outputs

- Mealy or Moore?
**Microcoding: Ver 1.0**

<table>
<thead>
<tr>
<th>state label</th>
<th>cntrl flow</th>
<th>conditional targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF₁</td>
<td>next</td>
<td>R/I-type LW SW Br Jump</td>
</tr>
<tr>
<td>IF₂</td>
<td>next</td>
<td></td>
</tr>
<tr>
<td>IF₃</td>
<td>next</td>
<td></td>
</tr>
<tr>
<td>IF₄</td>
<td>branch</td>
<td>ID ID ID IF₁</td>
</tr>
<tr>
<td>ID</td>
<td>next</td>
<td></td>
</tr>
<tr>
<td>EX₁</td>
<td>next</td>
<td></td>
</tr>
<tr>
<td>EX₂</td>
<td>branch</td>
<td>WB MEM₁ MEM₁ IF₁</td>
</tr>
<tr>
<td>MEM₁</td>
<td>next</td>
<td></td>
</tr>
<tr>
<td>MEM₂</td>
<td>next</td>
<td></td>
</tr>
<tr>
<td>MEM₃</td>
<td>next</td>
<td></td>
</tr>
<tr>
<td>MEM₄</td>
<td>next</td>
<td>WB IF₁ IF₁ IF₁ IF₁</td>
</tr>
<tr>
<td>WB</td>
<td>branch</td>
<td>IF₁ IF₁ IF₁ IF₁ IF₁</td>
</tr>
</tbody>
</table>

A systematic approach to FSM sequencing/control

**Microcontroller**

- A stripped-down processor for sequencing and control
  - control states are like µPC
  - µPC indexed into a µprogram ROM to select an µinstruction
  - well-formed control-flow architecture
  - fields in the µinstruction maps to control signals
  - Why not also support µprogram-visible states and ALU ops?

- Very elaborate mcontrollers have been built
  - some µcontrollers had full-fledged ISAs
  - µISAs for CISC machines in many ways inspired RISC ISA
Performance Analysis

- \( T_{\text{wall-clock}} = \text{No.Instructions} \times CPI \times T_{\text{clk}} \)
  - "No.Instruction" is fixed for a given ISA and application mix

- For a fixed ISA and application mix it is meaningful to compare
  \[ T_{\text{avg-inst}} = CPI \times T_{\text{clk}} \]
  or
  \[ \text{MIPS} = \text{IPC} \times f_{\text{clk}} \]
  where:
  - million instructions per second
  - instructions per cycle
  - frequency in MHz

- Single-Cycle Implementation
  \( 1 \times 1,667\text{MHz} = 1667 \text{ MIPS} \)

- Multi-Cycle Implementation
  \[ \text{IPC}_{\text{avg}} \times 20,000 \text{ MHz} = 2235 \text{ MIPS} \]

- Assume: 25% LW, 10% SW, 45% ALU, 15% Branch and 5% Jumps
  - weighted arithmetic mean of CPI \( \Rightarrow 8.95 \)
  - weighted harmonic mean of IPC \( \Rightarrow 0.1117 \)
Reducing Datapath by Resource Reuse

How to reuse the same adder for both additions in the same instruction.

Previous example of reuse by mutually exclusive conditions

Reducing Datapath by Sequential Reuse

to IR or not to IR?
Removing Redundancies

- Could also reduce down to a single register read-write port!
- Do we want to remove all redundancies?

Control Points
New Sequential Control Signals

<table>
<thead>
<tr>
<th>signal</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUSrcB[1:0]</td>
<td>00  2nd ALU input from 2nd RF read port (latched in B)</td>
</tr>
<tr>
<td></td>
<td>01  2nd ALU input is 4 (for PC increment)</td>
</tr>
<tr>
<td></td>
<td>10  2nd ALU input is sign-extended “IR[ 15:0 ]”</td>
</tr>
<tr>
<td></td>
<td>11  2nd ALU input is sign-extended “IR[ 15:0 ],00”</td>
</tr>
<tr>
<td>PCSrc[1:0]</td>
<td>00  next PC from ALU</td>
</tr>
<tr>
<td></td>
<td>01  next PC from ALUOut</td>
</tr>
<tr>
<td></td>
<td>10  next PC from IR (jump target)</td>
</tr>
</tbody>
</table>

When both PCWrite and PCWriteCond are de-asserted, PC latching is disabled
Old Control Signals
(similar to single-cycle)

<table>
<thead>
<tr>
<th></th>
<th>When De-asserted</th>
<th>When asserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDest</td>
<td>RF write select according to IR[20:16]</td>
<td>RF write select according to IR[15:11]</td>
</tr>
<tr>
<td>RegWrite</td>
<td>RF write disabled</td>
<td>RF write enabled</td>
</tr>
<tr>
<td>MemRead</td>
<td>Memory read disabled</td>
<td>Memory read port return load value</td>
</tr>
<tr>
<td>MemWrite</td>
<td>Memory write disabled</td>
<td>Memory write enabled</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>Steer ALU result (latched in ALUOut) to RF write port</td>
<td>Steer memory load result (latched in MDR) to RF write port</td>
</tr>
</tbody>
</table>

Synchronous Register Transfers

- Synchronous state with latch enables
  - PC, IR, RF, MEM
- Synchronous state that always latch
  - A, B, ALUOut, MDR
- One can enumerate all possible combinational "register transfers" in the datapath
- For example starting from PC
  - IR ← MEM[PC ]
    - requires IRWrite=1, IorD=0, MemRead=1
  - PC ← PC,JumpTarget
    - requires PCWrite=1, PCSource=2'b10
  - PC ← PC+ALUSrcB
  - MDR ← MEM[PC ]
  - ALUOut ← PC + ALUSrcB
  ......
Useful Register Transfers

- PC ← PC+4
- PC ← ALUOut (if PCWriteCond is asserted)
- PC ← PC[ 31:28 ],IR[ 25:0 ],2'b00
- IR ← MEM[ PC ]
- A ← RF[ IR[ 25:21 ] ]
- B ← RF[ IR[ 20:16 ] ]
- ALUOut ← A + B
- ALUOut ← A + sign-extend (IR[ 15:0 ])
- ALUOut ← PC + ( sign-extend (IR[ 15:0 ]) << 2 )
- MDR ← MEM[ ALUOut ]
- MEM[ ALUOut ] ← B
- RF[ IR[ 15:11 ] ] ← ALUOut,
- RF[ IR[ 20:16 ] ] ← ALUOut

RT Sequencing: R-Type ALU

- IF
  - IR ← MEM[ PC ]
  - PC ← PC+4
- ID
  - A ← RF[ IR[ 25:21 ] ]
  - B ← RF[ IR[ 20:16 ] ]
- EX
  - ALUOut ← A + B
- MEM
  -
- WB

if MEM[PC] == ADD rd rs rt
gpr[rd] ← gpr[rs] + gpr[rt]  
PC ← PC + 4
Can utilize each resource only once per control step (cycle)
RT Sequencing: LW

- **IF**
  - IR ← MEM(PC)
  - PC ← PC+4

- **ID**
  - A ← RF(IR[25:21])
  - B ← RF(IR[20:16])

- **EX**
  - ALUOut ← A + sign-extend(IR[15:0])

- **MEM**
  - MDR ← MEM(ALUOut)

- **WB**
  - RF(IR[20:16]) ← MDR

if MEM(PC) == LW rt offset16 (base)
EA = sign-extend(offset) + GPR[base]
GPR[rt] ← MEM[translate(EA)]
PC ← PC + 4

RT Sequencing: Branch

- **IF**
  - IR ← MEM(PC)
  - PC ← PC+4

- **ID**
  - A ← RF(IR[25:21])
  - B ← RF(IR[20:16])
  - ALUOut ← PC + (sign-extend(IR[15:0]) << 2)

- **EX**
  - PC ← ALUOut (only if condition is met)

- **MEM**

- **WB**

if MEM(PC) == BEQ rs rt immediate16
  target = PC + sign-extend(immediate) x 4 +4
  if GPR[rs] == GPR[rt] then
    PC ← target
  else
    PC ← PC + 4
Combined RT Sequencing

<table>
<thead>
<tr>
<th>R-Type</th>
<th>LW</th>
<th>SW</th>
<th>Branch</th>
<th>Jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>start:</td>
<td>IR ← MEM[PC]</td>
<td>PC ← PC+4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>A ← RF[IR[25:21]]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B ← RF[IR[20:16]]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALUOut ← PC + (sign-extend(IR[15:0]) &lt;&lt;2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>next</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALUOut ← A + B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ALUOut ← A + sign-extend(IR[15:0])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC ← ALUOut</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opcode dependent steps</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PC ← PC[31:28]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>go to IR[25:0], goto</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>start</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RF[IR[15:11]] ← MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MEM[ALUOut] ← PC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>goto</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>start</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RF[IR[15:11]] ← MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MEM[ALUOut] ← PC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>goto</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>start</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RTs in each state corresponds to some setting of the control signals.

Horizontal Microcode

[Diagram showing a microcode controller with a microcode storage, microprogram counter, adder, address select logic, and control signals for ALUSrcA, IorD, IRWrite, PCWrite, PCWriteCond, etc.]

Control Store: \(2^n \times k\) bit (not including sequencing)
**Vertical Microcode**

1-bit signal means do this RT (or combination of RTs)

If done right (i.e., m<<n, and m<<k), two ROMs together (2^n x m + 2^n x k bit) should be smaller than horizontal microcode ROM (2^n x k bit)

**Microcoding for CISC**

- Can we extend the \( \mu \)controller and datapath
  - to support a new instruction I haven’t thought of yet
  - to support a complex instruction, e.g. polyf

- Yes, and probably more
  - if I can sequence an arbitrary RISC instruction then I can sequence an arbitrary “RISC program” as a \( \mu \)program sequence
  - will need some \( \mu \)ISA state (e.g. loop counters) for more elaborate \( \mu \)programs
  - more elaborate \( \mu \)ISA features also make life easier

- \( \mu \)coding allows very simple datapath do very powerful computation
  - a datapath as simple as a Turning machine is universal
  - \( \mu \)code enables a minimal datapath to emulate any ISA you like (with a commensurate slow down)
Nanocode and Millicode

- **Nanocode**
  - a level below µcode
  - µprogrammed control for sub-systems (e.g., a complicated floating-point module) that acts as a slave in a µcontrolled datapath
  - e.g., the polyf sequence may be generated by a separate nanocontroller in the FPU

- **Millicode**
  - a level above µcode
  - ISA-level subroutines hardcoded into a ROM that can be called by the µcontroller to handle really complicated operations
  - e.g., to add polyf to MIPS, one may code up polyf as a software routine that is called by the µcontroller when the polyf opcode is decoded

- In both cases, we avoid complicating the main µcontroller with polyf support. The power of abstractions!!

Nanocode Concept Illustrated

A "mcoded" processor implementation

We refer to this as "nanocode" when a mcoded subsystem is embedded in a mcoded system.