Chapter 5
The Processor: Datapath & Control

Dr. Izadi
We're ready to look at an implementation of the MIPS

Simplified to contain only:
- memory-reference instructions: lw, sw
- arithmetic-logical instructions: add, sub, and, or, slt
- control flow instructions: beq, j

Generic Implementation:
- use the program counter (PC) to supply instruction address
- get the instruction from memory
- read (write) from (to) registers
- the op-code determines exactly what to do

All instructions use the ALU after reading the registers
More Implementation Details

- Abstract / Simplified View:

- Two types of functional units:
  - Elements that operate on data values (combinational)
    - ALU
  - Elements that contain state (sequential)
    - Registers and memory
State Elements

- Unclocked vs. Clocked
- Clocks used in synchronous logic
  - when should an element that contains state be updated?

![Diagram showing clock period, rising edge, and falling edge](image)

- Clock period
- Rising edge
- Falling edge
An Unclocked State Element

The set-reset latch

Output depends on present inputs and also on past inputs

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

R

Q

S

Q

Q
D-latch

- Two inputs:
  - the data value to be stored (D)
  - the clock signal (C) indicating when to read & store D

- Two outputs:
  - the value of the internal state (Q) and its complement
Latches and Flip-flops

- Output is equal to the stored value inside the element (don't need to ask for permission to look at the value)
- Change of state (value) is based on the clock
  - Latches: whenever the clock is asserted (high or low) and the input changes
  - Flip-flop: whenever the clock is asserted (low to high or high to low transition) and the input changes (edge-triggered methodology)
- Note:
  - "logically true" could mean electrically low
D flip-flop

- Output changes only on the clock edge
Our Implementation

- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements
Clocking Methodology

- A clocking methodology defines when signals can be read and written
  - wouldn't want to read a signal at the same time it was being written

- All storage elements are clocked by the same clock edge

- Cycle Time
  - CLK-to-Q + Longest Delay Path + Setup + Clock Skew
  - (CLK-to-Q + Shortest Delay Path - Clock Skew) > Hold Time
Abstraction

- Make sure you understand the abstractions!
- Sometimes it is easy to think you do, when you don’t
Register File

- Read operation using D flip-flops and MUX’s

What is the function of “Mux” above?
Register File – Read and Write

- How many registers can we read and write at the same time?
- Does this support MIPS instructions requirements?
Building the Datapath

- Include the functional units we need for each instruction
- Use multiplexors to stitch them together
Datapath for Fetch and R-Instructions

- Portion of datapath for fetching instructions and updating PC

- R-format instructions: read two registers and write one register

**ALU Control:**
- 000  AND
- 001  OR
- 010  add
- 110  subtract
- 111  set-on-less-than
Datapath for \textit{lw} and \textit{sw} Instructions

\texttt{lw \$t1, offset\_value($t2)} \text{ or } \texttt{sw \$t1, offset\_value ($t2)}

- Compute memory address
- Sign extend 16 bit to 32 bit
Building \textit{beq} Instruction

\textbf{beq $t1, t2, offset}$

- Compare registers, use ALU to affect Z flag
- If not equal, next PC $<=$ PC +4
- If equal, sign extend the offset and shift by two
A Simple Implementation of a Datapath

- Covers: lw, sw, beq, add, sub, and, or, set-on-less-than
- Use multiplexors to stitch them together
Implementation of the Datapath
Three Instruction Classes

• R-Type Instruction

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

rd: destination

• Load and Store Instructions

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit offset</th>
</tr>
</thead>
</table>

rt: destination

• Branch Instruction

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit offset</th>
</tr>
</thead>
</table>

rt: destination
Completed Data Path

- **PC**: Program Counter
- **MemtoReg**: Memory to Register
- **RegWrite**: Register Write
- **ALUSrc**: ALU Source
- **MemWrite**: Memory Write
- **MemRead**: Memory Read
- **ALU Control**: ALU Control
- **RegDst**: Register Destination
- **Instruction memory**: Memory for instructions
- **Add**: Adder
- **Shift left 2**: Shift left by 2 bits
- **32**: 32-bit result
- **16**: 16-bit result
- **Sign extend**: Sign extension
- **Zero**: Zero extension
- **ALU result**: ALU result
- **Add**: Add
- **4**: Constant 4
Control

- Using the op-code from the instruction, the control issues signals to:
  - Selecting the operations to perform (ALU, read/write, etc.)
  - Controlling the flow of data (multiplexer inputs)
- ALU's operation based on instruction type and function code
- Example: what should the ALU do with the instruction

```
add $8, $17, $18
```

<table>
<thead>
<tr>
<th>000000</th>
<th>10001</th>
<th>10010</th>
<th>01000</th>
<th>00000</th>
<th>100000</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
</tr>
</tbody>
</table>
Control

- Example: what should the ALU do with the instruction

\[ \text{lw } $1, 100($2) \]

<table>
<thead>
<tr>
<th>35</th>
<th>2</th>
<th>1</th>
<th>100</th>
</tr>
</thead>
</table>

\[ \text{op \hspace{1cm} rs \hspace{1cm} rt \hspace{1cm} 16 \text{ bit offset}} \]

*Why is the ALU code for subtract is 110 and not 011?*
**ALU Control**

- Must describe hardware to compute 3-bit ALU control input
  - Given instruction type
    - 00 = lw, sw
    - 01 = beq
    - 11 = arithmetic
  - Function code for arithmetic

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>ALUOp</th>
<th>F5</th>
<th>F4</th>
<th>F3</th>
<th>F2</th>
<th>F1</th>
<th>F0</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>010</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>110</td>
</tr>
<tr>
<td>1</td>
<td>1 (X)</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>1 (X)</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td>1</td>
<td>1 (X)</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>1 (X)</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>1</td>
<td>1 (X)</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>111</td>
</tr>
</tbody>
</table>

| ALU Control: |
|-----|-----|
| 000 | AND |
| 001 | OR  |
| 010 | add |
| 110 | subtract |
| 111 | set-on-less-than |
ALU Control

Muli-level decoding can reduce size of control unit and increase its speed.

<table>
<thead>
<tr>
<th>ALUOp</th>
<th>Funct field</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X X X X X X</td>
<td>010 add</td>
</tr>
<tr>
<td>0</td>
<td>X X X X X X</td>
<td>110 sub</td>
</tr>
<tr>
<td>1</td>
<td>X X 0 0 0 0</td>
<td>010 add</td>
</tr>
<tr>
<td>1</td>
<td>X X 0 0 1 0</td>
<td>110 sub</td>
</tr>
<tr>
<td>1</td>
<td>X X 0 1 0 0</td>
<td>000 and</td>
</tr>
<tr>
<td>1</td>
<td>X X 0 1 0 1</td>
<td>001 or</td>
</tr>
<tr>
<td>1</td>
<td>X X 1 0 1 0</td>
<td>111 slt</td>
</tr>
</tbody>
</table>
A Simple Control
Control

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>00 0000</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>10 0011</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>10 1011</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>00 0100</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Single Cycle Control

- Simple combinational logic

![Diagram showing ALU control block and logic gates for R-format operations]

- ALUOp1
- ALUOp0
- Operation0
- Operation1
- Operation2

- Outputs: RegDst, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, ALUOp1, ALUOp0
Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path
Single Cycle Implementation

- Calculate cycle time assuming negligible delays except: memory (2ns), ALU and adders (2ns), register file access (1ns)
Single Cycle Implementation

Calculate cycle time assuming negligible delays except:
- memory (2ns), ALU and adders (2ns), register file access (1ns)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instr. Memory</th>
<th>Register Read</th>
<th>ALU Op.</th>
<th>Data Memory</th>
<th>Reg. Write</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>6 ns</td>
</tr>
<tr>
<td>lw</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8 ns</td>
</tr>
<tr>
<td>sw</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td>7 ns</td>
</tr>
<tr>
<td>beq</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>5 ns</td>
</tr>
</tbody>
</table>
Single Cycle Problems:

- Cycle time should accommodate the longest instruction.
  - What if we had more complicated instructions like floating point?
- Can use a unit only once during a cycle
  - May need multiple copies of some functional units (wasteful of area)
- One Solution:
  - use a “smaller” cycle time
  - have different instructions take different numbers of cycles
  - a “multi-cycle” datapath:
Multi-cycle Approach

- We will be reusing functional units
  - ALU used to compute address and the new PC value
  - Same memory used for instruction and data
- Break up the instructions into steps, each step takes a cycle
  - Balance the amount of work to be done
  - Restrict each cycle to use only one major functional unit
- Our control signals will not be determined solely by instruction, but also by the current step
  - e.g., what should the ALU do for an “add” instruction?
- At the end of a cycle (step)
  - Store values for use in later steps
  - Introduce additional “internal” registers
Multi-cycle Approach

- **Added components**
  - IR and MDR both needed during the same cycle
  - A and B registers to hold operand values
  - ALUOut register
Multicycle Approach

- Internal registers except IR are updated every clock cycle; No write control

- Need to add new MUX’s and expand existing MUX’s
Instructions from ISA Perspective

- Consider each instruction from perspective of ISA.
- Example:
  - The add instruction changes a register.
  - Register specified by bits 15:11 of instruction.
  - Instruction specified by the PC.
  - New value is the sum ("op") of two registers.
  - Registers specified by bits 25:21 and 20:16 of the instruction

\[
\text{Reg}[\text{Memory}[\text{PC}][15:11]] \leq \text{Reg}[\text{Memory}[\text{PC}][25:21]] \text{ op } \text{Reg}[\text{Memory}[\text{PC}][20:16]]
\]
- In order to accomplish this we must break up the instruction.
  (kind of like introducing variables when programming)
Breaking Down of an Instruction

- ISA definition of arithmetic:
  \[
  \text{Reg}[\text{Memory}[\text{PC}][15:11]] \leq \text{Reg}[\text{Memory}[\text{PC}][25:21]] \text{ op } \text{Reg}[\text{Memory}[\text{PC}][20:16]]
  \]

- Could break down to:
  - IR \leq \text{Memory}[\text{PC}]
  - A \leq \text{Reg}[\text{IR}[25:21]]
  - B \leq \text{Reg}[\text{IR}[20:16]]
  - ALUOut \leq A \text{ op } B
  - Reg[IR[15:11]] \leq \text{ALUOut}

- We forgot an important part of the definition of arithmetic!
  - PC \leq PC + 4
Idea Behind Multicycle Approach

- We define each instruction from the ISA perspective (do this!)
- Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)
- Introduce new registers as needed (e.g., A, B, ALUOut, MDR, etc.)
- Finally try and pack as much work into each step (avoid unnecessary cycles) while also trying to share steps where possible (minimizes control, helps to simplify solution)
- Result: Our book’s multicycle Implementation!
Five Execution Steps

1. Instruction fetch
2. Instruction decode and register fetch
3. Execution, memory address computation, or branch completion
4. Memory access or R-type instruction completion
5. Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!
Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC using RTL "Register-Transfer Language"
  
  \[
  IR \leftarrow \text{Memory}[\text{PC}]; \quad \text{PC} \leftarrow \text{PC} + 4;
  \]

MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt (in case we need them) and put them in A & B
- Compute the branch address in case the instruction is a branch; put it in ALUOut
  \[
  A \leftarrow \text{Reg}[IR[25-21]]; \\
  B \leftarrow \text{Reg}[IR[20-16]]; \\
  \text{ALUOut} \leftarrow PC + (\text{sign-extend}(\text{IR}[15-0]) \ll 2);
  \]

ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00

- We are Looking at the instruction and determine what to do in the next cycle
- We aren't setting any control lines based on the instruction type
Step 3: Instruction Dependent

- ALU is performing one of three functions, based on instruction type
- (ignore j instruction for now)

- Memory Reference: (lw or sw)
  ALUOut <= A + sign-extend(IR[15-0]);

- R-type:
  ALUOut <= A op B;

- Branch: beq
  if (A==B) PC <= ALUOut

(compute address)

ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00

(execute)

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
  go to step 4

(complete branch)

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01

lw  sw

end of execution
fetch next instruction

fetch next instruction
For beq

For lw and sw
### Step 4: R-type or Memory Access

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>R-type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lw</strong></td>
<td>MDR = Memory[ALUOut]; MemRead</td>
<td>Reg[IR[15-11]] = ALUOut;</td>
</tr>
<tr>
<td></td>
<td><em>(read from memory)</em></td>
<td><em>(store result)</em></td>
</tr>
<tr>
<td><strong>sw</strong></td>
<td>Memory[ALUOut] = B; MemWrite</td>
<td>RegDst = 1</td>
</tr>
<tr>
<td></td>
<td><em>(write to memory)</em></td>
<td>REGWrite</td>
</tr>
<tr>
<td></td>
<td>IorD = 1</td>
<td>MemtoReg = 0</td>
</tr>
<tr>
<td></td>
<td><em>end of execution</em></td>
<td><em>end of execution</em></td>
</tr>
<tr>
<td></td>
<td>fetch next instruction</td>
<td></td>
</tr>
</tbody>
</table>
PCSource

ECE 45432  SUNY New Paltz

46
**Step 5 – Write Back**

- Only for lw instructions to store data read from memory into a register

Reg[IR[20-16]]= MDR;

RegDest = 0
RegWrite
MemtoReg = 1
## Control Summary

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>IR = Memory[PC]</td>
<td>PC = PC + 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction decode/</td>
<td>A = Reg [IR[25-21]]</td>
<td>B = Reg [IR[20-16]]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>register fetch</td>
<td></td>
<td>ALUOut = PC + (sign-extend (IR[15-0]) &lt;&lt; 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A + sign-extend (IR[15-0])</td>
<td>if (A == B) then PC = ALUOut</td>
<td>PC = PC [31-28] H (IR[25-0]&lt;&lt;2)</td>
</tr>
<tr>
<td>computation, branch/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jump completion</td>
<td>Memory access or R-type</td>
<td>Load: MDR = Memory[ALUOut] or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td>Reg [IR[15-11]] = ALUOut</td>
<td>Store: Memory [ALUOut] = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory read completion</td>
<td>Load: Reg[IR[20-16]] = MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed

- Use the information we’ve accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming

- Implementation can be derived from specification
Review: Finite State Machines

- Finite state machines:
  - A set of states and
  - Next state function (determined by current state and the input)
  - Output function (determined by current state and possibly input)

- We’ll use a Moore machine (output based only on current state)
- How does Moore machine various from Mealy machine?
Review: finite state machines

Example:
A friend would like you to build an “electronic eye” for use as a fake security device. The device consists of three lights lined up in a row, controlled by the outputs Left, Middle, and Right, which, if asserted, indicate that a light should be on. Only one light is on at a time, and the light “moves” from left to right and then from right to left, thus scaring away thieves who believe that the device is monitoring their activity. Draw the graphical representation for the finite state machine used to specify the electronic eye. Note that the rate of the eye’s movement will be controlled by the clock speed (which should not be too great) and that there are essentially no inputs.
Graphical Specification of FSM

- don’t care if not mentioned
- asserted if name only
- otherwise exact value

How many state bits will we need?
Simple Questions

How many cycles will it take to execute this code?

```assembly
lw $t2, 0($t3)
lw $t3, 4($t3)
beq $t2, $t3, Label    #assume not
add $t5, $t2, $t3
sw $t5, 8($t3)
Label: ...```

What is going on during the 8th cycle of execution?

In what cycle does the actual addition of $t2 and $t3 takes place?
Multiple Cycle Datapath

- Ideal Memory
- RAdr
- WrAdr
- Din
- Dout
- Instruction Reg
- IRWr
- RegDst
- RegWr
- ALUOp
- Control
- IorD
- MemWr
- ALUSelA
- Mux
- Rs
- Rt
- Rd
- Ra
- Rw
- busA
- busW
- busB
- Mux
- ExtOp
- MemtoReg
- ALUSelB
- ALUOut
- ALU
- Zero
- Target
- BrWr
- PCSrc
- PCWr
- PCWrCond
- Zero
- PC
- 0
- PCWrCond
- 0
- 1
- 5
- 0
- 1
- 2
- 3
- 4
- << 2
- Extend
- Imm
- 16
- 32
- 32
- 32
- 32
- 32
- 32
- 32
- 32
- 32
Finite State Machine for Control

Implementation:

- Control logic
- Instruction register
- State register
- Outputs: PCWrite, PCWriteCond, IorD, MemRead, MemWrite, IRWrite, MemtoReg, PCSource, ALUOp, ALUSrcB, ALUSrcA, RegWrite, RegDst, NS3, NS2, NS1, NS0
If I picked a horizontal or vertical line could you explain it?
ROM Implementation

- ROM = "Read Only Memory"
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is m-bits, we can address $2^m$ entries in the ROM.
  - our outputs are the bits of data that the address points to.

- $m$ is the "height", and $n$ is the "width"
ROM Implementation

- How many inputs are there?
  6 bits for opcode, 4 bits for state = 10 address lines
  (i.e., $2^{10} = 1024$ different addresses)

- How many outputs are there?
  16 datapath-control outputs, 4 state bits = 20 outputs

- ROM is $2^{10} \times 20 = 20K$ bits (and a rather unusual size)

- Rather wasteful, since for lots of the entries, the outputs are the same
  - i.e., opcode is often ignored
ROM vs PLA

- Break up the table into two parts
  - 4 state bits tell you the 16 outputs, $2^4 \times 16$ bits of ROM
  - 10 bits tell you the 4 next state bits, $2^{10} \times 4$ bits of ROM
  - Total: 4.3K bits of ROM

- PLA is much smaller
  - Can share product terms
  - Only need entries that produce an active output
  - Can take into account don't cares

- Size: (#inputs´ #product-terms) + (#outputs´ #product-terms)
  - For this example = (10x17)+(20x17) = 510 PLA cells
- PLA cells usually about the size of a ROM cell (slightly bigger)
Another Implementation Style

- Complex instructions
  - The "next state" is often current state + 1
Microprogramming

- Control is the hard part of processor design
  - Datapath is fairly regular and well-organized
  - Memory is highly regular
  - Control is irregular and global

- Microprogramming:
  - A Particular Strategy for Implementing the Control Unit of a processor by "programming" at the level of register transfer operations

- Microarchitecture:
  - Logical structure and functional capabilities of the hardware as seen by the microprogrammer

- Historical Note:
  - IBM 360 Series first to distinguish between architecture & organization
    Same instruction set across wide range of implementations, each with different cost/performance
Macroinstruction Versus Microinstruction

Main Memory

User program plus Data
this can change!

one of these is mapped into one of these at RTL level

ADD
SUB
AND

DATA

CPU

AND microsequence

e.g., Fetch
Fetch Operand(s)
Calculate
Save Answer(s)

execution unit
The state diagrams that arise define the controller for an instruction set processor are highly structured.

Use this structure to construct a simple “microsequencer”.

Control reduces to programming this very simple device:
- microprogramming.
Microprogramming
Implementation of the Control
# Microprogramming

- **microinstruction**: low level control instruction which defines a set of datapath control signals.

- **A specification methodology**
  - appropriate if hundreds of opcodes, modes, cycles, etc.
  - signals specified symbolically using microinstructions

## Table

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU control</th>
<th>SRC1</th>
<th>SRC2</th>
<th>Register control</th>
<th>Memory</th>
<th>PCWrite control</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Add</td>
<td>PC</td>
<td>4</td>
<td>Read PC</td>
<td>ALU</td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Add</td>
<td>PC</td>
<td>Extshft</td>
<td>Read</td>
<td></td>
<td>Dispatch 1</td>
<td></td>
</tr>
<tr>
<td>Mem1</td>
<td>Add</td>
<td>A</td>
<td>Extend</td>
<td>Read ALU</td>
<td></td>
<td>Dispatch 2</td>
<td></td>
</tr>
<tr>
<td>LW2</td>
<td></td>
<td></td>
<td></td>
<td>Write MDR</td>
<td>Read ALU</td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write ALU</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>SW2</td>
<td>Func code</td>
<td>A</td>
<td>B</td>
<td>Write ALU</td>
<td></td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td>Rformat1</td>
<td></td>
<td></td>
<td></td>
<td>Write ALU</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>BEQ1</td>
<td>Subt</td>
<td>A</td>
<td>B</td>
<td>ALUOut-cond</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>JUMP1</td>
<td></td>
<td></td>
<td></td>
<td>Jump address</td>
<td></td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>Field name</td>
<td>Value</td>
<td>Signals active</td>
<td>Comment</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>
<td></td>
</tr>
<tr>
<td><strong>ALU control</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>ALUOp = 00</td>
<td></td>
<td>Cause the ALU to add.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subt</td>
<td>ALUOp = 01</td>
<td></td>
<td>Cause the ALU to subtract; this implements the compare for branches.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Func code</td>
<td>ALUOp = 10</td>
<td></td>
<td>Use the instruction's function code to determine ALU control.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SRC1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>ALUSrcA = 0</td>
<td></td>
<td>Use the PC as the first ALU input.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>ALUSrcA = 1</td>
<td></td>
<td>Register A is the first ALU input.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SRC2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>ALUSrcB = 00</td>
<td></td>
<td>Register B is the second ALU input.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>ALUSrcB = 01</td>
<td></td>
<td>Use 4 as the second ALU input.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extend</td>
<td>ALUSrcB = 10</td>
<td></td>
<td>Use output of the sign extension unit as the second ALU input.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extend</td>
<td>ALUSrcB = 11</td>
<td></td>
<td>Use the output of the shift-by-two unit as the second ALU input.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write ALU</td>
<td>RegWrite,</td>
<td></td>
<td>Write a register using the rd field of the IR as the register number and</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RegDst = 1,</td>
<td></td>
<td>the contents of the ALUOut as the data.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MemtoReg = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write MDR</td>
<td>RegWrite,</td>
<td></td>
<td>Write a register using the rt field of the IR as the register number and</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RegDst = 0,</td>
<td></td>
<td>the contents of the MDR as the data.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MemtoReg = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read PC</td>
<td>MemRead,</td>
<td></td>
<td>Read memory using the PC as address; write result into IR (and the MDR).</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lorD = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read ALU</td>
<td>MemRead,</td>
<td></td>
<td>Read memory using the ALUOut as address; write result into MDR.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lorD = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read ALU</td>
<td>MemWrite,</td>
<td></td>
<td>Write memory using the ALUOut as address, contents of B as the data.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lorD = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write ALU</td>
<td>MemWriteCond,</td>
<td></td>
<td>Write the output of the ALU into the PC.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>lorD = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>PCSource = 00</td>
<td></td>
<td>Write the output of the ALU into the PC.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCSource = 01</td>
<td></td>
<td>If the Zero output of the ALU is active, write the PC with the contents</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCSource = 10</td>
<td></td>
<td>Write the PC with the jump address from the instruction.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCSource = 01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PC write control</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUOut-cond</td>
<td>PCSource = 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCSource = 01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCSource = 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PCSource = 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq</td>
<td>AddrCt = 11</td>
<td></td>
<td>Choose the next microinstruction sequentially.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sequencing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetch</td>
<td>AddrCt = 00</td>
<td></td>
<td>Go to the first microinstruction to begin a new instruction.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dispatch 1</td>
<td>AddrCt = 01</td>
<td></td>
<td>Dispatch using the ROM 1.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dispatch 2</td>
<td>AddrCt = 10</td>
<td></td>
<td>Dispatch using the ROM 2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Details

#### Dispatch ROM 1

<table>
<thead>
<tr>
<th>Op</th>
<th>Opcode name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>R-format</td>
<td>0110</td>
</tr>
<tr>
<td>000010</td>
<td>jmp</td>
<td>1001</td>
</tr>
<tr>
<td>000100</td>
<td>beq</td>
<td>1000</td>
</tr>
<tr>
<td>100011</td>
<td>lw</td>
<td>0010</td>
</tr>
<tr>
<td>101011</td>
<td>sw</td>
<td>0010</td>
</tr>
</tbody>
</table>

#### Dispatch ROM 2

<table>
<thead>
<tr>
<th>Op</th>
<th>Opcode name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100011</td>
<td>lw</td>
<td>0011</td>
</tr>
<tr>
<td>101011</td>
<td>sw</td>
<td>0101</td>
</tr>
</tbody>
</table>

### State Number, Address-Control Action, Value of AddrCtl

<table>
<thead>
<tr>
<th>State number</th>
<th>Address-control action</th>
<th>Value of AddrCtl</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>Use dispatch ROM 1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Use dispatch ROM 2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Use incremented state</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Replace state number by 0</td>
<td>0</td>
</tr>
</tbody>
</table>
Maximally vs. Minimally Encoded

- No encoding:
  - 1 bit for each datapath operation
  - faster, requires more memory (logic)
  - used for Vax 780 — an astonishing 400K of memory!

- Lots of encoding:
  - send the microinstructions through logic to get control signals
  - uses less memory, slower

- Historical context of CISC:
  - Too much logic to put on a single chip with everything else
  - Use a ROM (or even RAM) to hold the microcode
  - It’s easy to add new instructions
Designing a Microinstruction Set

1. Start with a list of control signals.
2. Group signals together that make sense (vs. random): called “fields”.
3. Place fields in some logical order e.g., ALU operation & ALU operands first and microinstruction sequencing last.
4. Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals.
   - Use computers to design computers.
5. To minimize the width, encode operations that will never be used at the same time.
### Possible Design Paths of Control

<table>
<thead>
<tr>
<th>Initial Representation</th>
<th>Finite State Diagram</th>
<th>Microprogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequencing Control</td>
<td>Explicit Next State Function</td>
<td>Microprogram counter + Dispatch ROMs</td>
</tr>
<tr>
<td>Logic Representation</td>
<td>Logic Equations</td>
<td>Truth Tables</td>
</tr>
<tr>
<td>Implementation Technique</td>
<td>PLA</td>
<td>ROM</td>
</tr>
<tr>
<td></td>
<td>“hardwired control”</td>
<td>“microprogrammed control”</td>
</tr>
</tbody>
</table>
Microprogramming

Pros and Cons

- Ease of design
- Flexibility
  - Easy to adapt to changes in organization, timing, technology
  - Can make changes late in design cycle, or even in the field
- Can implement very powerful instruction sets (just more control memory)
- Generality
  - Can implement multiple instruction sets on same machine.
  - Can tailor instruction set to application.
- Compatibility
  - Many organizations, same instruction set
- Costly to implement
- Slow
Microcode: Trade-offs

- Distinction between specification and implementation is blurred

- Specification Advantages:
  - Easy to design and write
  - Design architecture and microcode in parallel

- Implementation (off-chip ROM) Advantages
  - Easy to change since values are in memory
  - Can emulate other architectures
  - Can make use of internal registers

- Implementation Disadvantages, SLOWER now that:
  - Control is implemented on same chip as processor
  - ROM is no longer faster than RAM
  - No need to go back and make changes
Historical Perspective

- In the ‘60s and ‘70s microprogramming was very important for implementing machines.
- This led to more sophisticated ISAs and the VAX.
- In the ‘80s RISC processors based on pipelining became popular.
- Pipelining the microinstructions is also possible!
- Implementations of IA-32 architecture since 486:
  - “hardwired control” for simpler instructions
    (few cycles, FSM control implemented using PLA or random logic)
  - “microcoded control” for more complex instructions
    (large numbers of cycles, central control store)
- The IA-64 architecture uses a RISC-style ISA and can be implemented without a large central control store.
Pentium 4

- Pipelining is important (last IA-32 without it was 80386 in 1985)
Somewhere in all that “control we must handle complex instructions

- Processor executes simple microinstructions, 70 bits wide (hardwired)
- 120 control lines for integer datapath (400 for floating point)
- If an instruction requires more than 4 microinstructions to implement, control from microcode ROM (8000 microinstructions)
- It’s complicated!
Summary

- If we understand the instructions, we can build a simple processor!
- If instructions take different amounts of time, multi-cycle is better
- Datapath implemented using:
  - Combinational logic for arithmetic logic unit
  - State holding elements for registers and memory
- Control implemented using:
  - Combinational logic for single-cycle implementation
  - Finite state machine for multi-cycle implementation