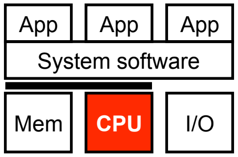
CIS 371
Computer Organization and Design

Unit 2: Single-Cycle Datapath and Control

This Unit: Single-Cycle Datapaths



- Digital logic basics
  - Focus on useful components
- Mapping an ISA to a datapath
  - MIPS example
- Single-cycle control

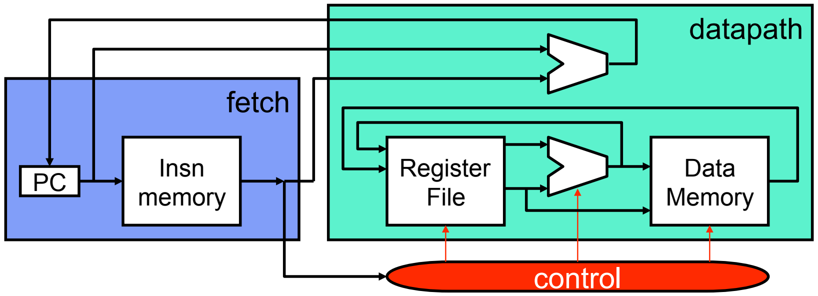
Readings

- Digital logic
  - P&H, Appendix C (on CD)
- Basic datapath
  - P&H, Chapter 4.1 – 4.4 (well-written, relates to lecture well)

So You Have an ISA…

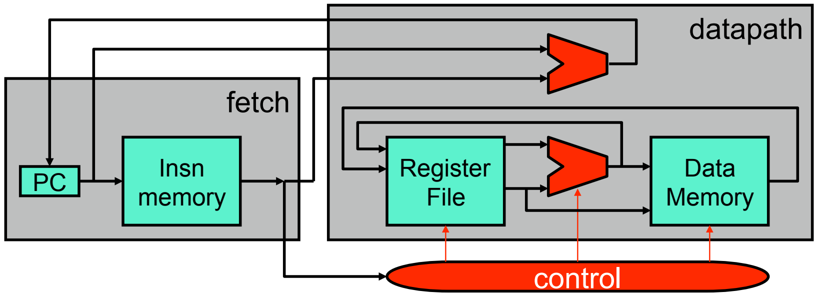
- … not useful without a piece of hardware to execute it

Implementing an ISA



- **Datapath**: performs computation (registers, ALUs, etc.)
  - ISA specific: can implement every insn (single-cycle: in one pass!)
- **Control**: determines which computation is performed
  - Routes data through datapath (which regs, which ALU op)
- **Fetch**: get insn, translate opcode into control
- **Fetch** → **Decode** → **Execute** “cycle”

Two Types of Components

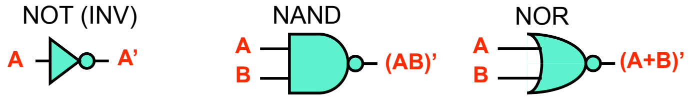


- **Purely combinational**: stateless computation
  - ALUs, muxes, control
  - Arbitrary Boolean functions
- **Combinational/sequential**: storage
  - PC, insn/data memories, register file
  - Internally contain some combinational components

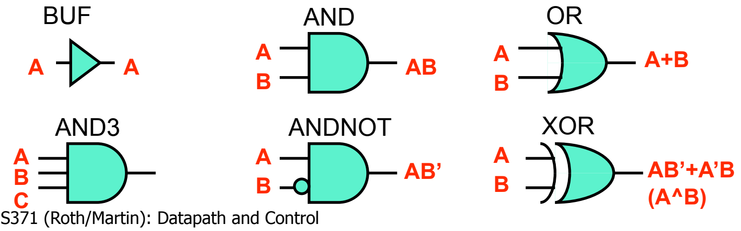
Digital Logic Review

Building Blocks: Logic Gates

- **Logic gates:** implement Boolean functions
  - Basic gates: NOT, NAND, NOR
    - Underlying CMOS transistors are naturally inverting (○ = NOT)



  - NAND, NOR are “Boolean complete”



Boolean Functions and Truth Tables

- Any Boolean function can be represented as a truth table
  - **Truth table**: point-wise input → output mapping
  - Function is disjunction of all rows in which "Out" is 1

```
A,B,C → Out
0,0,0 → 0
0,0,1 → 0
0,1,0 → 0
0,1,1 → 0
1,0,0 → 0
1,0,1 → 1
1,1,0 → 1
1,1,1 → 1
```

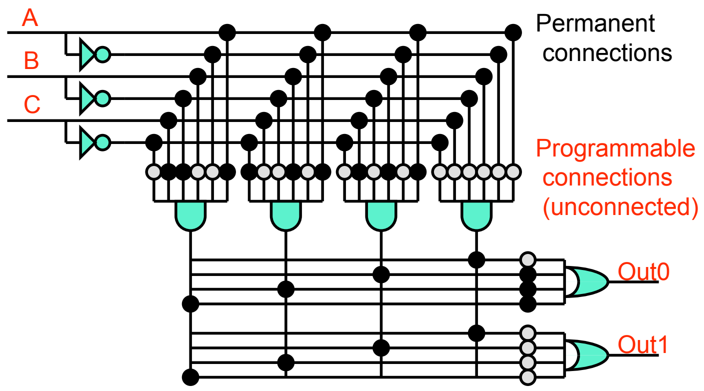
  - Example above: Out = AB′C + ABC′ + ABC

Truth Tables and PLAs

- Implement Boolean function by implementing its truth table
  - Takes two levels of logic
    - Assumes inputs and inverses of inputs are available (usually are)
  - First level: ANDs (product terms)
  - Second level: ORs (sums of product terms)
- **PLA (programmable logic array)**
  - Flexible circuit for doing this

PLA Example

- PLA with 3 inputs, 2 outputs, and 4 product terms
  - Out0 = AB′C + ABC′ + ABC



Boolean Algebra

- **Boolean Algebra**: rules for rewriting Boolean functions
  - Useful for simplifying Boolean functions
    - Simplifying = reducing gate count, reducing gate "levels"
  - Rules: similar to logic (0/1 = F/T)
    - **Identity**: A1 = A, A+0 = A
    - **0/1**: A0 = 0, A+1 = 1
    - **Inverses**: (A′)′ = A
    - **Idempotency**: AA = A, A+A = A
    - **Tautology**: AA′ = 0, A+A′ = 1
    - **Commutativity**: AB = BA, A+B = B+A
    - **Associativity**: A(BC) = (AB)C, A+(B+C) = (A+B)+C
    - **Distributivity**: A(B+C) = AB+AC, A+(BC) = (A+B)(A+C)
    - **DeMorgan's**: (AB)′ = A′+B′, (A+B)′ = A′B′

Logic Minimization

- **Logic minimization**
  - Iterative application of rules to reduce function to simplest form
  - There are tools for automatically doing this
  - Example below: function from slide #8

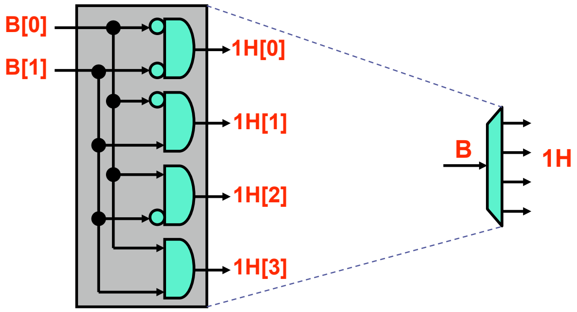
```
Out = AB'C + ABC' + ABC
Out = A(B'C + BC' + BC)       // distributivity
Out = A(B'C + (BC' + BC))     // associativity
Out = A(B'C + B(C'+C))        // distributivity (on B)
Out = A(B'C + B1)             // tautology
Out = A(B'C + B)              // 0/1
Out = A((B'+B)(C+B))          // distributivity (on +B)
Out = A(1(B+C))               // tautology
Out = A(B+C)                  // 0/1
```

Non-Arbitrary Boolean Functions

- PLAs implement Boolean functions point-wise
  - E.g., represent f(X) = X+5 as [0→5, 1→6, 2→7, 3→8, …]
  - Mainly useful for "arbitrary" functions, no compact representation
- Many useful Boolean functions are not arbitrary
  - Have a compact representation
  - E.g., represent f(X) = X+5 as X+5
  - Examples
    - Decoder
    - Multiplexer
    - Adder: e.g., X+5 (or more generally, X+Y)

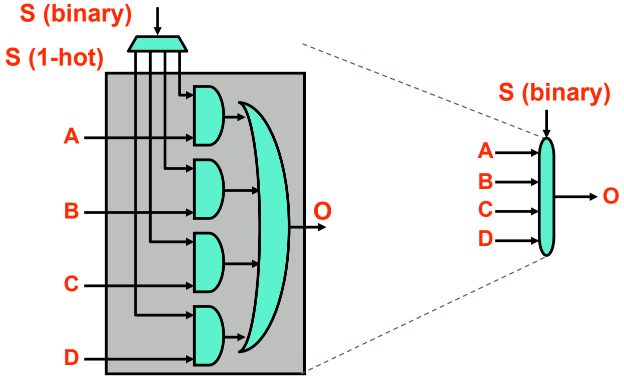
Decoder

- **Decoder**: converts binary integer to 1-hot representation
  - Binary representation of $0 \ldots 2^N-1$: N bits
  - 1 hot representation of $0 \ldots 2^N-1$: 2^N bits
    - J represented as Jth bit 1, all other bits zero
  - Example below: 2-to-4 decoder



Multiplexer (Mux)

- **Multiplexer (mux)**: selects output from N inputs
  - Example: 1-bit 4-to-1 mux
  - Not shown: N-bit 4-to-1 mux = N 1-bit 4-to-1 muxes + 1 decoder



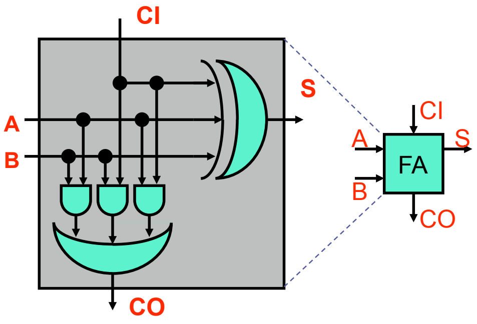
Adder

- **Adder**: adds/subtracts two 2C binary integers
  - **Half adder**: adds two 1-bit "integers", no carry-in
  - **Full adder**: adds three 1-bit "integers", includes carry-in
  - **Ripple-carry adder**: N chained full adders add 2 N-bit integers
  - **To subtract**: negate B input, set bit 0 carry-in to 1

Full Adder

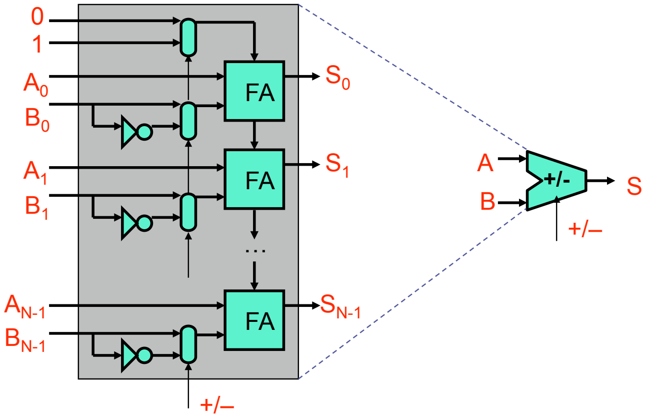
- What is the logic for a full adder?
  - Look at truth table

CI	A	B	→	CO	S
0	0	0	→	0	0
0	0	1	→	0	1
0	1	0	→	0	1
0	1	1	→	1	0
1	0	0	→	0	1
1	0	1	→	1	0
1	1	0	→	1	0
1	1	1	→	1	1



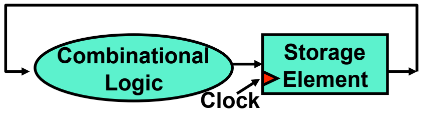
- **S = C'A'B + C'AB' + CA'B' + CAB = C ^ A ^ B**
- **CO = C'AB + CA'B + CAB' + CAB = CA + CB + AB**

N-bit Adder/Subtracter



- More later when we cover arithmetic

Sequential Logic & Synchronous Systems



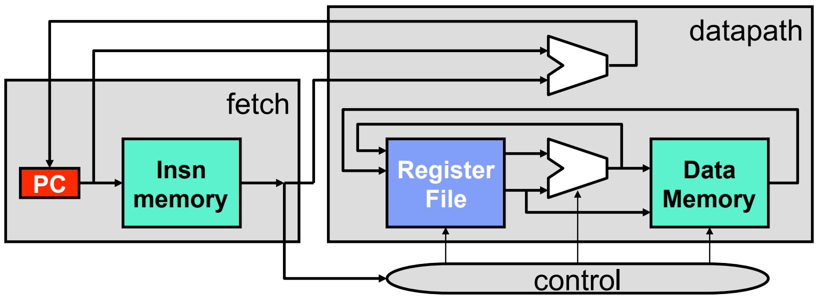
Combinational Logic

Storage Element

Clock

- Processors are complex fine state machines (FSMs)
  - Combinational (compute) blocks separated by storage elements
    - State storage: memories, registers, etc.
- **Synchronous systems**
  - **Clock**: global signal acts as write enable for all storage elements
    - Typically marked as triangle
  - All state elements write together, values move forward in lock-step
  - + Simplifies design: design combinational blocks independently
- Aside: asynchronous systems
  - Same thing, but … no clock
  - Values move forward using explicit handshaking
  - ± May have some advantages, but difficult to design

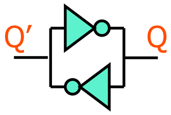
Datapath Storage Elements



- Three main types of storage elements
  - **Singleton registers**: PC
  - **Register files**: ISA registers
  - **Memories**: insn/data memory

Cross-Coupled Inverters (CCIs)

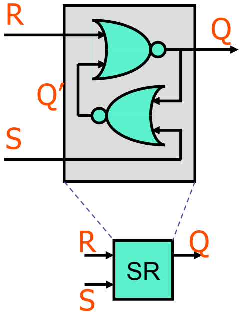
- **Cross-coupled inverters (CCIs)**
  - **Primitive "storage element" for storing state**
  - Most storage arrays (regfile, caches) implemented this way
  - Where is the input and where is the output?



S-R Latch

- **S-R (set-reset) latch**
  - Cross-coupled NOR gates
  - Distinct inputs/outputs

```
S,R → Q
0,0 → oldQ
0,1 → 0
1,0 → 1
1,1 → 0
```



  - S=0, R=0? circuit degenerates to cross-coupled INVs
  - S=1, R=1? not very useful
  - Not really used … except as component in something else

D Latch

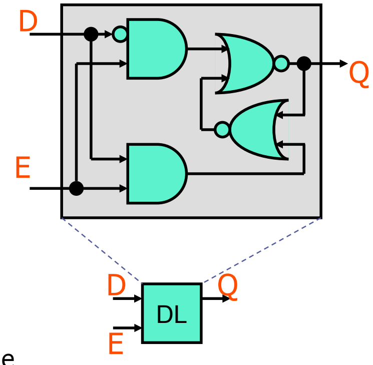
- **D latch**: S-R latch + …
  - control that makes S=R=1 impossible

```
E,D → Q
0,0 → oldQ
0,1 → oldQ
1,0 → 0
1,1 → 1
```

  - In other words

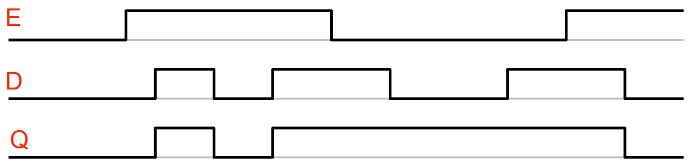
```
0,D → oldQ
1,D → D
```

  - In words
    - When E is 1, Q gets D
    - When E is 0, Q retains old value

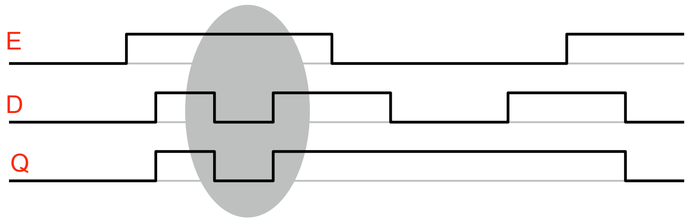


Timing Diagrams

- Voltage {0,1} diagrams for different nodes in system
  - "Digitally stylized": changes are vertical lines (instantaneous?)
  - Reality is analog, changes are continuous and smooth
- Timing diagram for a D latch



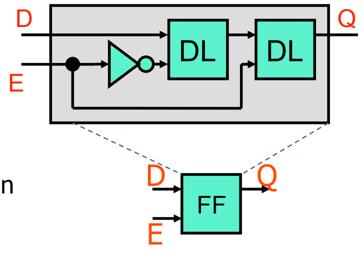
Triggering: Level vs. Edge

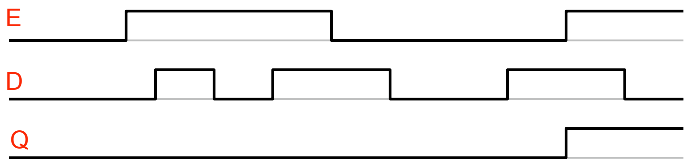


- The D-latch is **level-triggered**
  - The latch is open for writing as long as E is 1
  - If D changes continuously, so does Q
  - – May not be the functionality we want
- Often easier to reason about an **edge-triggered** latch
  - The latch is open for writing only on E transition (0 → 1 or 1 → 0)
  - + Don't need to worry about fluctuations in value of D

D Flip-Flop

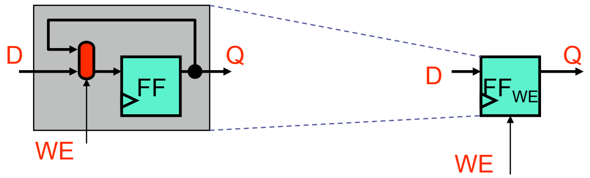
- **D Flip-Flop**: also called master-slave flip-flop
  - Sequential D-latches
  - Enabled by inverse signals
  - First latch open when E = 0
  - Second latch open when E = 1
  - Overall effect?
    - D FF latches D on 0→1 transition
  - E is the "clock" signal input





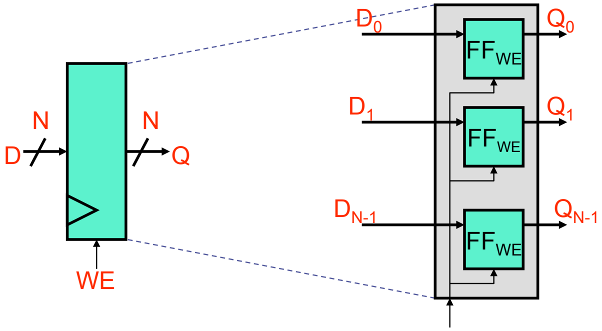
FF$_{WE}$: FF with Separate Write Enable

- **FF$_{WE}$**: FF with separate write enable
  - FF D(ata) input is MUX of D and Q, WE selects



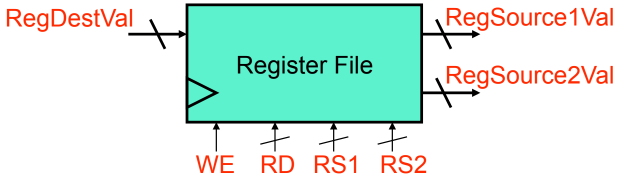
  - Alternative: FF E(nable) input is AND of CLK and WE
    - + Fewer gates
    - – Creates timing problems
      - ▪ Do not try to do logic on CLK in Verilog
      - ▪ No, really.

Singleton Register



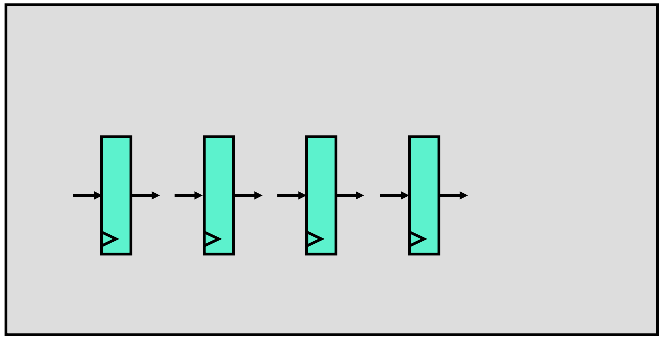
- **Register**: one *N*-bit storage word
  - Non-multiplexed input/output: data buses write/read same word
- Implementation: FF_{WE} array with shared write-enable (WE)
  - FFs written on CLK edge if WE is 1 (or if there is no WE)

Register File



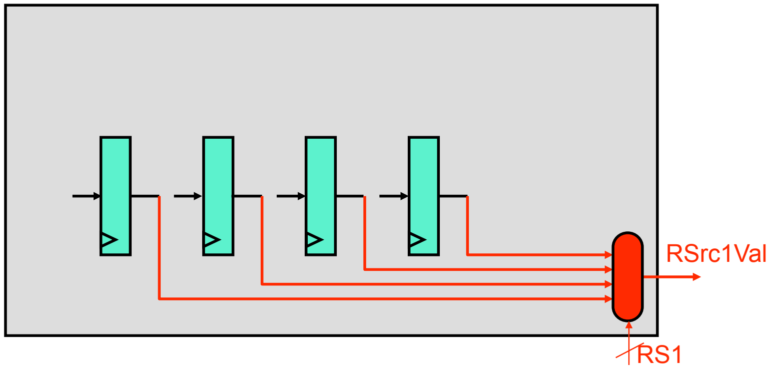
- **Register file**: M N-bit storage words
  - Multiplexed input/output: data buses write/read "random" word
- **"Port"**: set of buses for accessing a random word in array
  - Data bus (N-bits) + address bus (log$_2$M-bits) + optional WE bit
  - P ports = P parallel and independent accesses
- MIPS integer register file
  - 32 32-bit words, two read ports + one write port (why?)

Register File (Port) Implementation



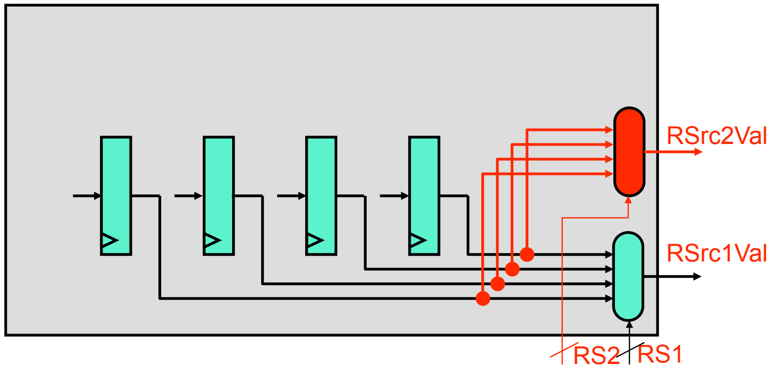
- Register file with four registers

Add a Read Port



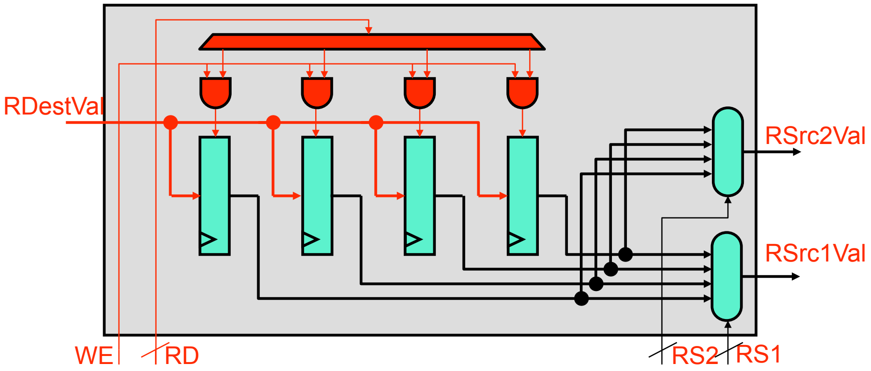
- Output of each register into 4to1 mux (RSrc1Val)
  - RS1 is select input of RSrc1Val mux

Add Another Read Port



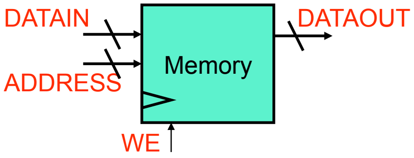
- Output of each register into another 4to1 mux (RSrc2Val)
  - RS2 is select input of RSrc2Val mux

Add a Write Port



- Input RegDestVal into each register
  - Enable only one register's WE: (Decoded RD) & (WE)
- What if we needed two write ports?

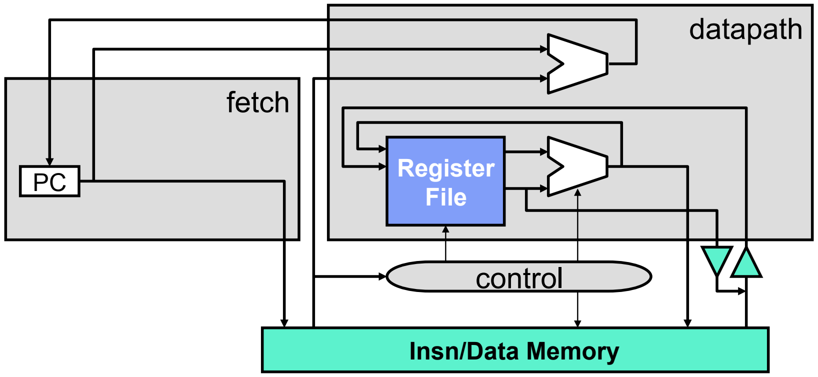
Another Useful Component: Memory



- Register file: M N-bit storage words
  - Few words (< 256), many ports, dedicated read and write ports
  - Synchronous
- **Memory**: M N-bit storage words, yet not a register file
  - Many words (> 1024), few ports (1, 2), shared read/write ports
- Leads to different implementation choices
  - Lots of circuit tricks and such

MIPS Datapath & Control

Unified vs Split Memory Architecture



- **Unified architecture**: unified insn/data memory
  - LC3, MIPS, every other ISA
- **Harvard architecture**: split insn/data memories
  - LC4

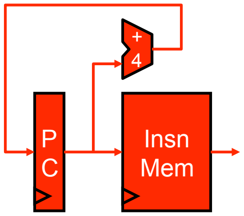
Datapath for MIPS ISA

- Registers in MIPS are $0, $2… $31
- Consider only the following instructions

```
add  $1,$2,$3        $1 = $2 + $3         (add)
addi $1,2,$3         $1 = 2 + $3          (add immed)
lw  $1,4($3)         $1 = Memory[4+$3]    (load)
sw  $1,4($3)         Memory[4+$3] = $1    (store)
beq $1,$2,PC_relative_target   (branch equal)
j absolute_target          (unconditional jump)
```

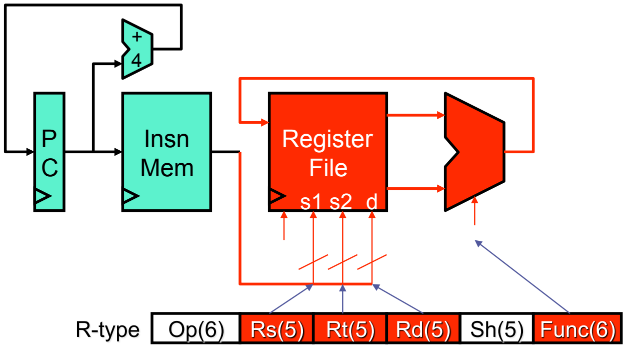
- Why only these?
  - Most other instructions are the same from datapath viewpoint
  - The one's that aren't are left for you to figure out

Start With Fetch



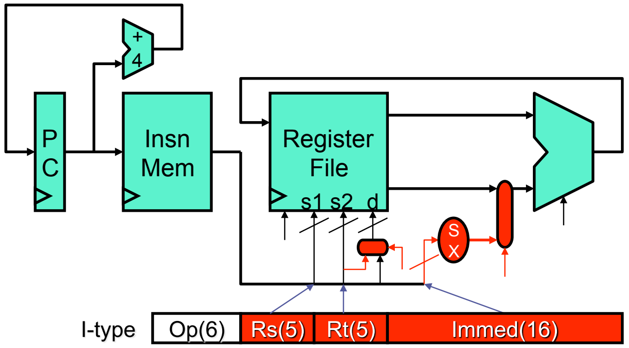
- PC and instruction memory (Harvard architecture, for now)
- A +4 incrementer computes default next instruction PC

First Instruction: add



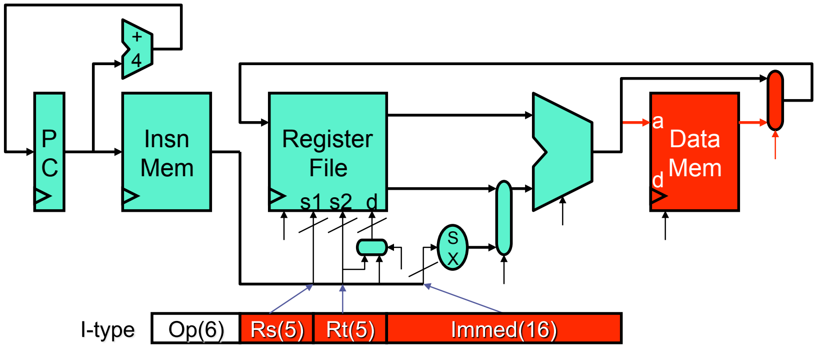
- Add register file and ALU

Second Instruction: **addi**



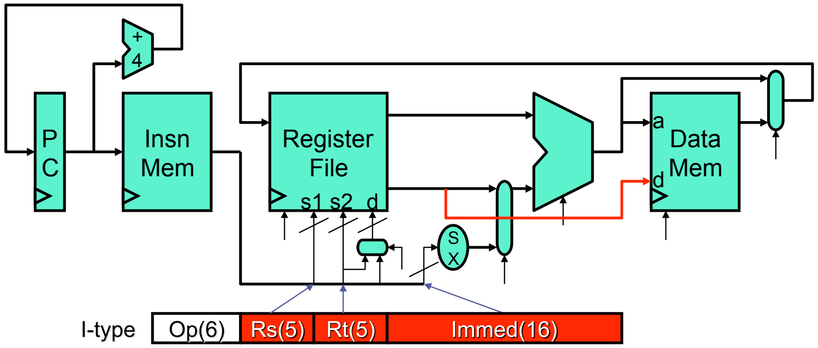
- Destination register can now be either Rd or Rt
- Add sign extension unit and mux into second ALU input

Third Instruction: **lw**



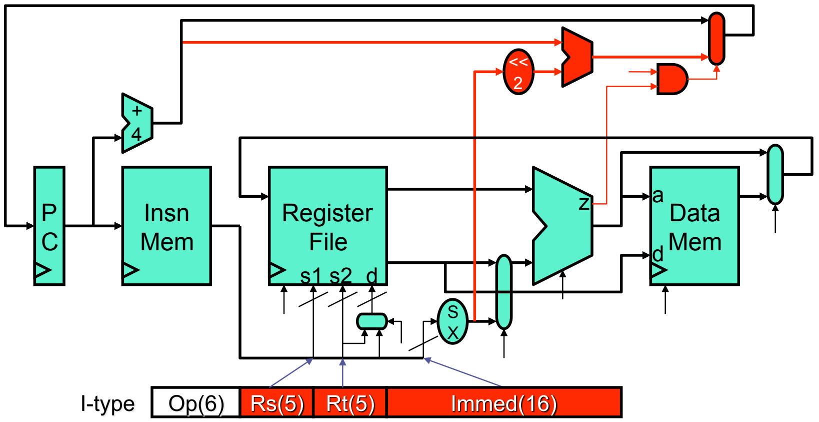
- Add data memory, address is ALU output
- Add register write data mux to select memory output or ALU output

Fourth Instruction: **sw**



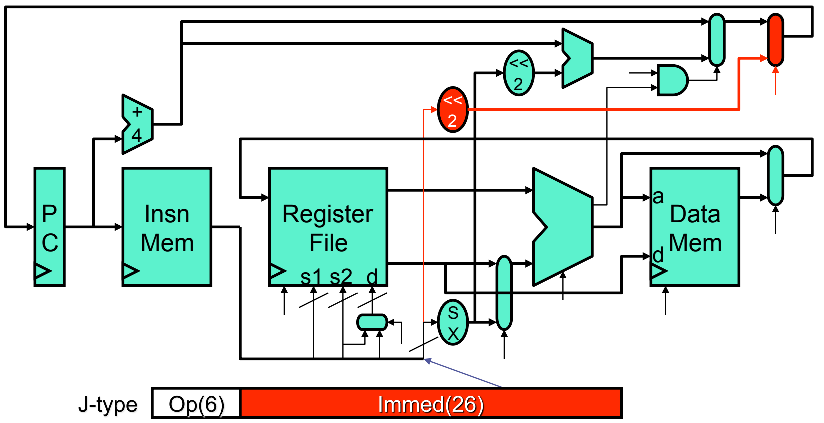
- Add path from second input register to data memory data input

Fifth Instruction: **beq**



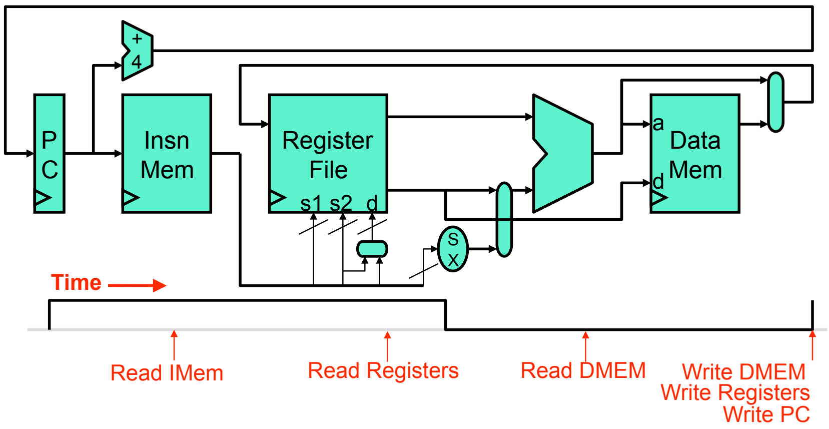
- Add left shift unit and adder to compute PC-relative branch target
- Add PC input mux to select PC+4 or branch target

Sixth Instruction: **j**



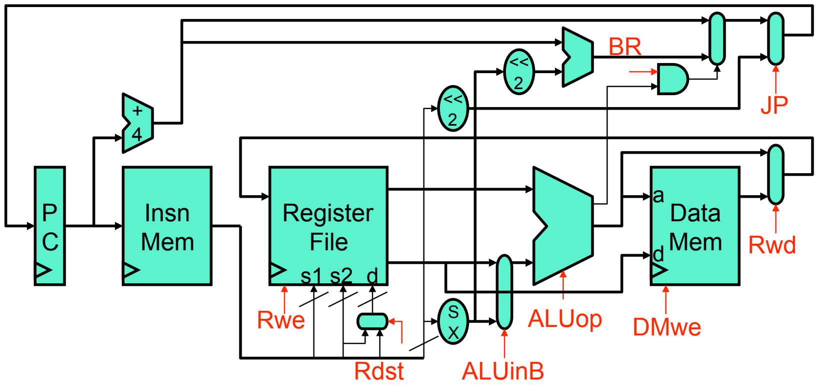
- Add shifter to compute left shift of 26-bit immediate
- Add additional PC input mux for jump target

"Continuous Read" Datapath Timing



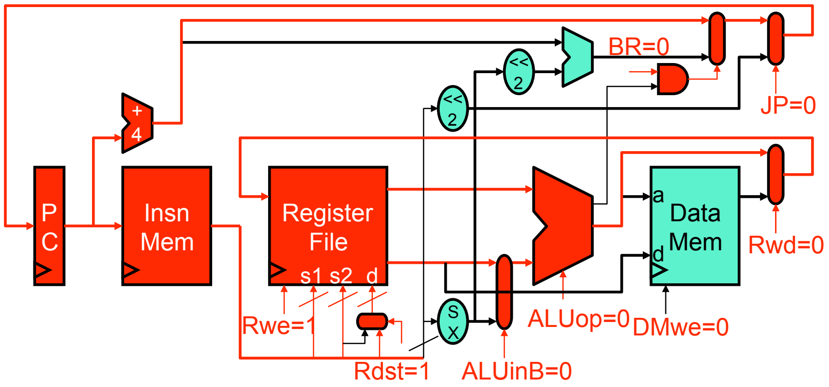
- Works because writes (PC, RegFile, DMem) are independent
- And because no read logically follows any write

What Is Control?

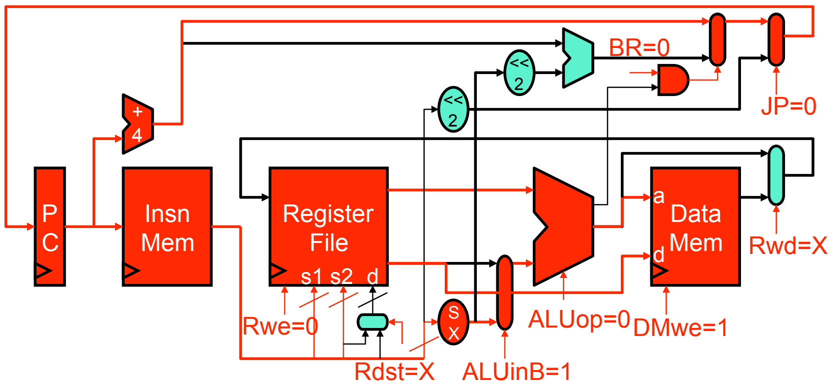


- 9 signals control flow of data through this datapath
  - MUX selectors, or register/memory write enable signals
  - A real datapath has 300-500 control signals

Example: Control for **add**

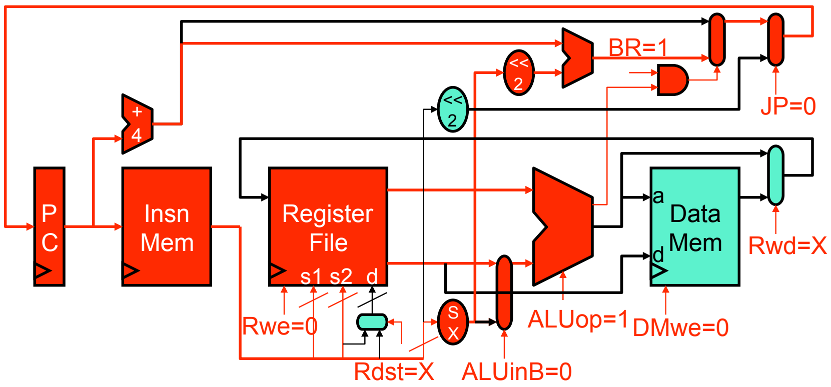


Example: Control for **sw**



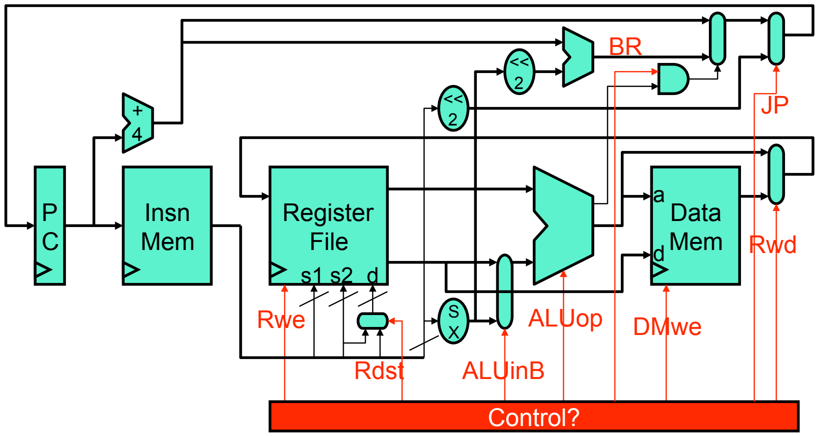
- Difference between `sw` and `add` is 5 signals
  - 3 if you don't count the X (don't care) signals

Example: Control for **beq**



- Difference between `sw` and `beq` is only 4 signals

How Is Control Implemented?

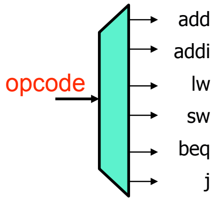


Implementing Control

- Each instruction has a unique set of control signals
  - Most are function of opcode
  - Some may be encoded in the instruction itself
    - E.g., the ALUop signal is some portion of the MIPS Func field
    - + Simplifies controller implementation
    - Requires careful ISA design

Control Implementation: ROM

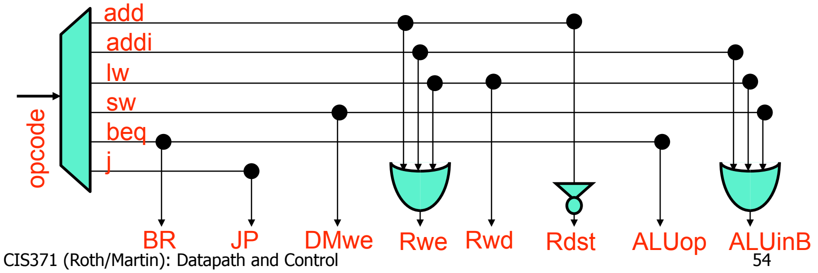
- **ROM (read only memory)**: like a RAM but unwritable
  - Bits in data words are control signals
  - Lines indexed by opcode
  - Example: ROM control for 6-insn MIPS datapath
  - X is "don't care"



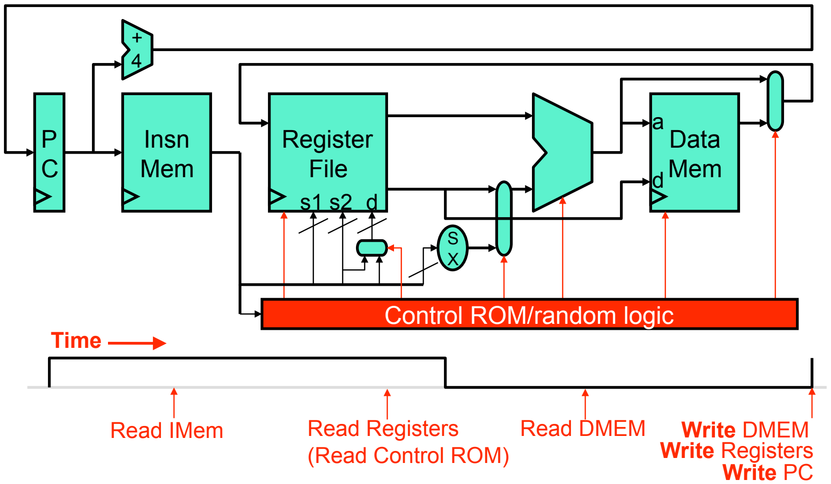
	BR	JP	ALUinB	ALUop	DMwe	Rwe	Rdst	Rwd
add	0	0	0	0	0	1	0	0
addi	0	0	1	0	0	1	1	0
lw	0	0	1	0	0	1	1	1
sw	0	0	1	0	1	0	X	X
beq	1	0	0	1	0	0	X	X
j	0	1	0	0	0	0	X	X

Control Implementation: Logic

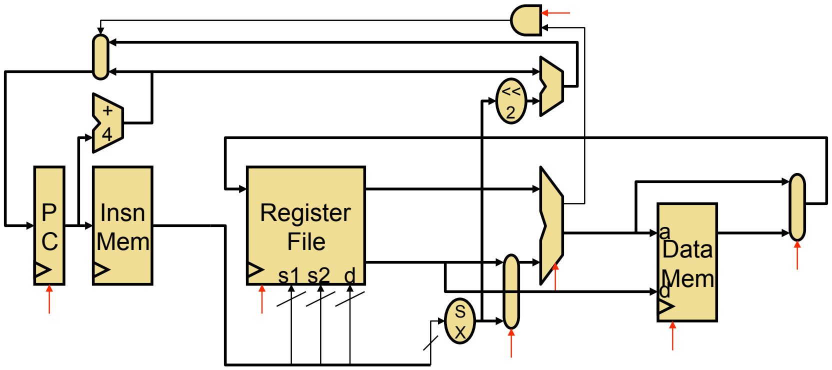
- Real machines have 100+ insns 300+ control signals
  - 30,000+ control bits (~4KB)
  - – Not huge, but hard to make faster than datapath (important!)
- Alternative: **logic gates** or "random logic" (unstructured)
  - Exploits the observation: many signals have few 1s or few 0s
  - Example: random logic control for 6-insn MIPS datapath



Datapath and Control Timing

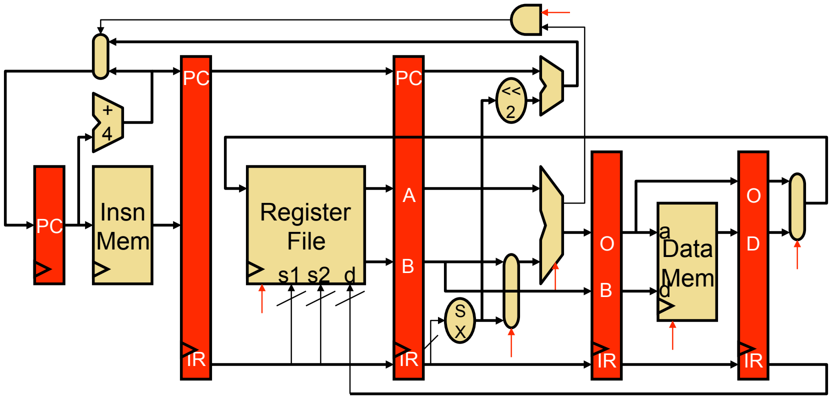


Single-Cycle Datapath Performance



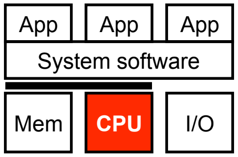
- One instruction per cycle (1 IPC or 1 CPI)
- Clock cycle time proportional to worst-case logic delay
  - In this datapath: insn fetch, decode, register read, ALU, data memory access, write register
  - Can we do better?

Foreshadowing: Pipelined Datapath



- Split datapath into multiple stages
  - Assembly line analogy
  - 5 stages results in up to 5x clock & performance improvement

Summary



App App App

System software

Mem CPU I/O

- Digital logic review
- Single-cycle datapath and control

- Next up:
  - Arithmetic
  - Performance & metrics