CSCI 2021: Basics of Hardware and CPU Architecture

Chris Kauffman

Last Updated:
Wed Nov 9 12:25:39 PM CST 2022

Logistics

Reading Bryant/O'Hallaron

Ch 4: Architectures

- Skimming is OK
- Lecture: high-level coverage

Ch 6: Memory (Next)

Goals

- Circuits that Compute
- Basics of Processor Arch
- Pipelining

Lab / HW 10

- Time C code using `time` command
- Observe strange results that raise questions
- Answer questions in lecture

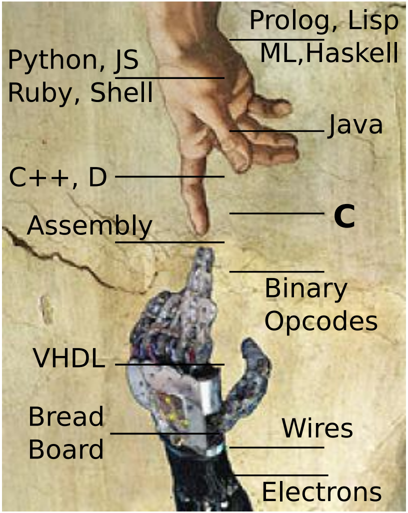
Project 4

- Post by Next Week
- Optimization + Performance
- Use knowledge of Arch / Memory to improve / explain execution speed

Machines that Compute

- Humans can perform algorithms, sadly **slow and error-prone**
- Want a machine which can do this faster with fewer errors
- Variety of machines have been built over time and technology to implement them has changed rapidly
- The following are high-level principles that haven't changed much

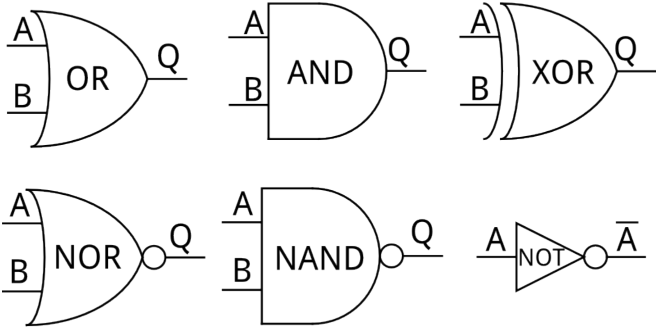
Pure Abstraction



Bare Metal

Logic Gates

- Abstract physical device that implements a boolean function
- May be implemented with a variety of components including **transistors**, vacuum tubes, mechanical devices, and water pressure
- Physical implementations have many trade-offs: cost, speed, difficulty to manufacture, miniaturization potential, wetness



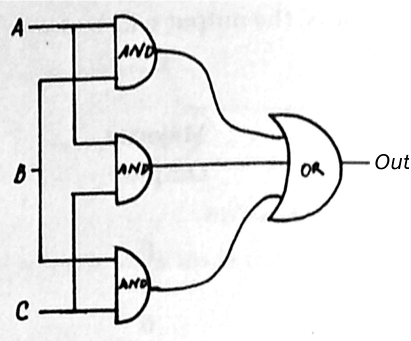
Combinatorial Circuits

- Combination of wires/gates with output solely dependent on inputs entering circuit
- No storage of information involved / **stateless**
- Distinguished from **sequential** circuits which necessarily introduce time and state
- Combinatorial circuits can compute any **Boolean Function** of inputs
  - Set inputs as 0/1
  - After a (short) delay, outputs are set
- Examples: AND, OR, NOT are obvious

Exercise: Example Combinatorial Circuit

Calculate the Truth Table for the circuit

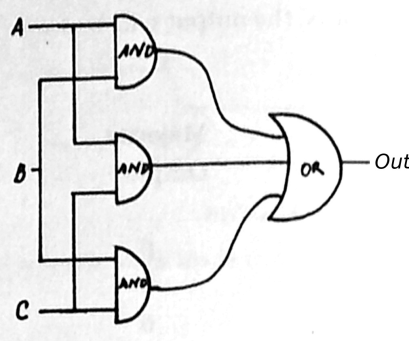
A	B	C	Out
0	0	0	?
0	0	1	?
0	1	0	?
0	1	1	?
1	0	0	?
1	0	1	?
1	1	0	?
1	1	1	?



- Speculate on the “meaning” of this circuit

Answer: Example Combinatorial Circuit

A	B	C	Out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



A "majority" circuit: Out is 1 when two or more of A,B,C are 1

Exercise: Comparing Majority-3 Circuits

- Both upper and lower circuits implement *Majority-3*: Same truth table
- Which is **better**?
- What criteria for "better" seems appropriate?

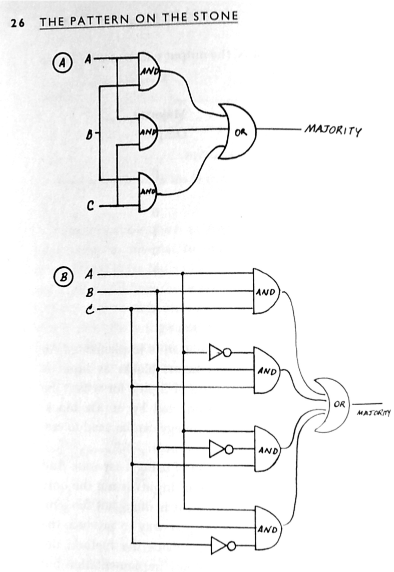


FIGURE 12

Answer: Comparing "Majority-3" Circuits

Criteria	Upper	Lower
Gate Kinds	2	3
Gate Count	4	8
Gate "Depth"	2	3
"Scalability"	Low	High

- "Scalability" is not well-defined, roughly how to "scale up" to majority 64
- Hardware designers spend time trying to design "better" circuits where "better" involves many criteria

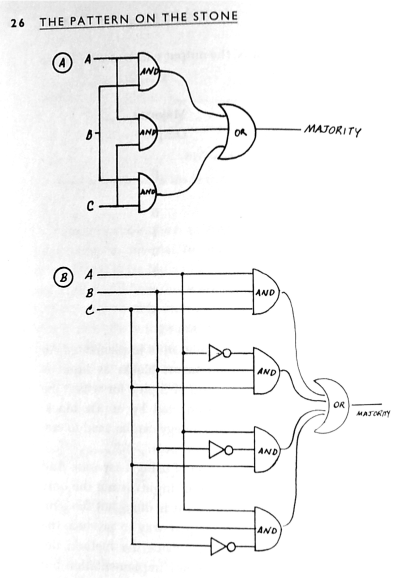
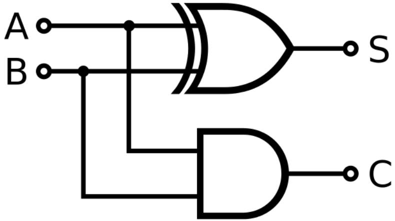


FIGURE 12

Adders

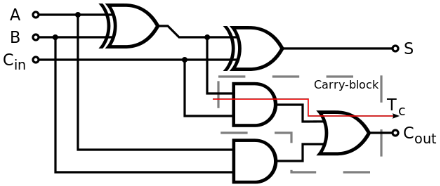
- Obviously want computers to add stuff
- An **adder** is a circuit that performs addition

1-bit Half Adder



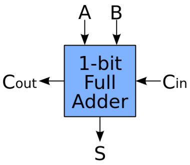
- "Adds" A and B
- S is the sum
- C is the carry
- Construct a **Truth Table** for the circuit

1-bit **Full** Adder

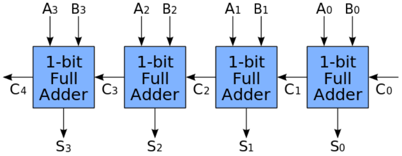


- "Adds" A, B, and C_{in}
- S is the sum
- C_{out} is the carry out
- Carry In/Out used to string adders together

Multi-bit Addition

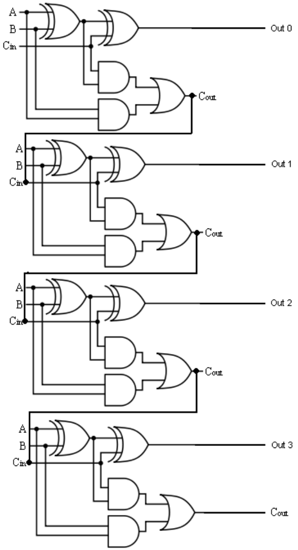


Combine 4 full adders to get a 4-bit ripple carry adder



Easily extends to 32- or 64-bit adders

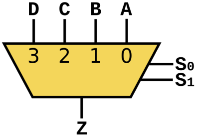
Full Gate Layout



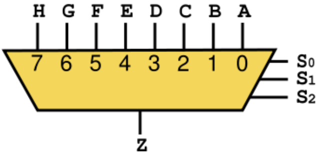
Multiplexers: MUX

- Used to "select" output from several inputs
- 2^N Inputs A,B,C,...
- N selection bits $S_{0,S1}$,...
- Output will be one of inputs "chosen" by selection bits
- Block diagram is a rectangle or trapezoid with inputs/outputs
- Will prove useful momentarily

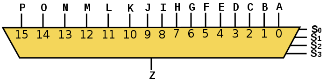
- 4-to-1 MUX



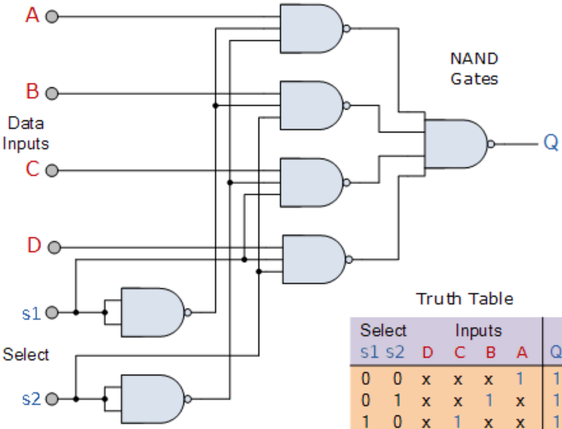
- 8-to-1 MUX



- 16-to-1 MUX

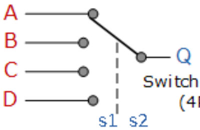


4-to-1 Multiplexer Circuit Diagram



Truth Table

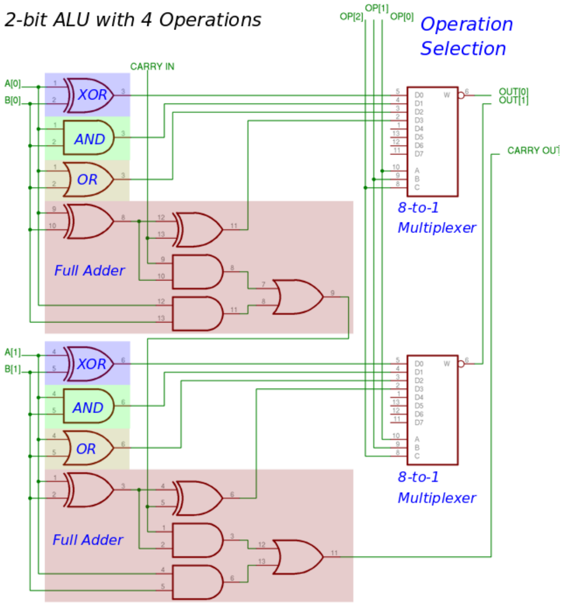
Select		Inputs				
s1	s2	D	C	B	A	Q
0	0	x	x	x	1	1
0	1	x	x	1	x	1
1	0	x	1	x	x	1
1	1	1	x	x	x	1



Switch Analogy
(4PST)

- Variety of ways to design a MUX
- One shown uses NAND gates exclusively
- Note output is true when selected input is true

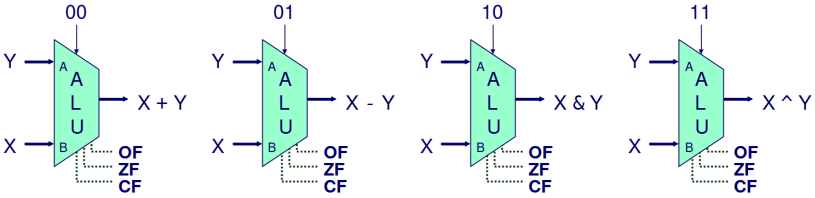
Arithmetic Logic Unit ALU: Select an Operation



- Combine some gates, an adder, and a MUX
- Start having something that looks useful
- Input for multiple ops like AND, OR, XOR, ADD are **simulataneously** computed
- Select an "operation" with selection bits, really just selecting which output to pass through

ALU and FLAGS

- Block diagram for ALUs are usually a wedge shape
- Along with arithmetic/logic, ALU usually produces condition codes which are among outputs from ALU
  - ZF: zero flag
  - OF: overflow flag
  - SF: sign flag
- Used in other parts of CPU for conditional jumps/moves



Hardware Design in the Old Days

- Hardware design originally done by hand
- Draw all the gates, transfer it to technical drawing material, peel, send, hope to heaven that nothing gets munged...
- Required tremendous discipline, still had bugs

Ted Jenkins remembers working on the first Intel product, the 3101 64-bit RAM. Actually, the first version was only a 63-bit RAM due to a simple error peeling one layer on the rubylith (drawing medium).[1]

[1] Andrew Volk, Peter Stoll, Paul Metrovich, "Recollections of Early Chip Development at Intel", Intel Technology Journal Q1, 2001

Modern Hardware Design: Specification Languages

- Modern design uses **hardware description languages**
- Verilog and VHDL pervasive, describe **behavior** of circuit
- *Synthesis:* convert description to gate layout with constraints like "use only NAND"
- *Verification:* simulate circuit to ensure correctness
- The invention of computers greatly accelerated development of better computers

VHDL for 4-bit ALU ^ & | +

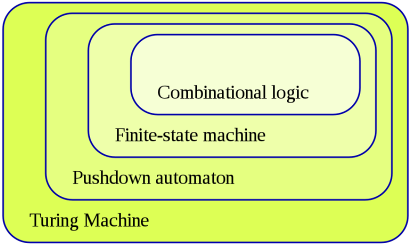
```
library IEEE;
entity alu is
 Port(A_IN : in signed(3 downto 0);
      B_IN : in signed(3 downto 0);
      OPER : in STD_LOGIC_VECTOR(1 downto 0);
      OUTP : out signed(3 downto 0));
end alu;

architecture Behavioral of alu is
begin
  process(A_IN, B_IN, OPER)
  begin
    case OPER is
      when "00" =>
        OUTP <= A_IN xor B_IN; --XOR gate
      when "01" =>
        OUTP <= A_IN and B_IN; --AND gate
      when "10" =>
        OUTP <= A_IN or  B_IN; --OR gate
      when "11" =>
        OUTP <= A_IN +   B_IN; --addition
    end case;

  end process;
end Behavioral;
```

Combinatorial vs Sequential Circuits

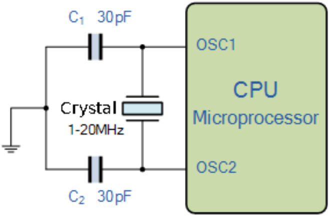
- Combinatorial circuits can do lots of things BUT don't constitute a complete programming system
- Need to represent **state**: store values, make future values depend on past state
- **Sequential circuits** introduce the notion of time and state to allow actual computation
- Most actual machines are state machines in some class like push-down automata or Turing machines (studied in 2011 and 4011)



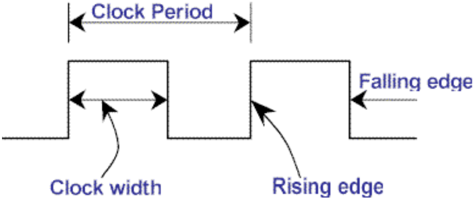
The class of problems that can be solved grows with more powerful machines.

Clock Circuits

- To move beyond combinatorial circuits, need a way to measure time
- A **Clock Circuit** does this
- Provides an oscillating signal of high/low voltages at a fixed frequency

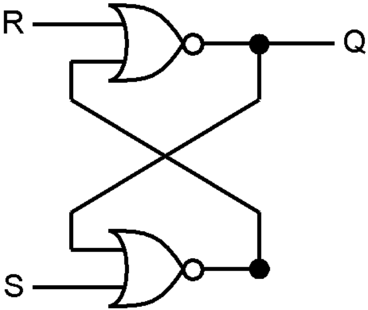


- Physical device: often **quartz crystal** which contracts when voltage is applied (*electrostriction*), expands when released
- Manufactured to have different periods/frequencies
- Circuitry attached to crystal causes oscillation at crystal's resonant frequency; circuitry can increase/decrease output frequency



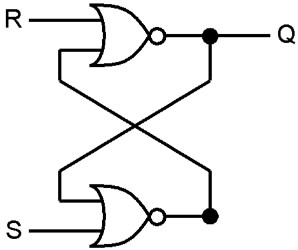
Examine: A Strange Circuit: SR Latch

- This one should bug you a little - why?
- Try computing a Truth Table for it...



Aswers: A Strange Circuit: SR Latch

- SR Latch uses **feedback to store one bit** which is output as Q
- Truth Tables less relevant than **State Transition Table**
- Shows what the next state will be based on previous state
- Inputs and Outputs
  - S is for "SET"
  - R is for "RESET"
  - Q is current stored value
  - Q_{next} is new stored value



State Transition Table

S	R	Q_{next}	Action
0	0	Q	hold state
0	1	0	reset
1	0	1	set
1	1	X	not allowed

Storage via Latches $\approx$ Flip-Flops

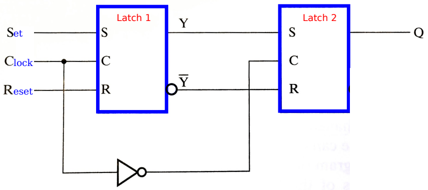
Specific combinations of latches yield the following nice properties

- Store a bit of information so long as power is supplied (not shown in diagrams)
- Constantly output the stored bit
- Change the bit on certain inputs
- **Only change stored bit** during the rising edge of an input signal - **the clock tick**
- Often referred to as a **Flip Flop**, commonly a *rising edge flip-flop*[2]
- Latches/Flip Flops can serve as a basis for **registers**

[2] There is no agreement on whether latches and flip-flops are the same or different so take care to understand context if going deeper. Relation above is adopted from some textbooks on digital design.

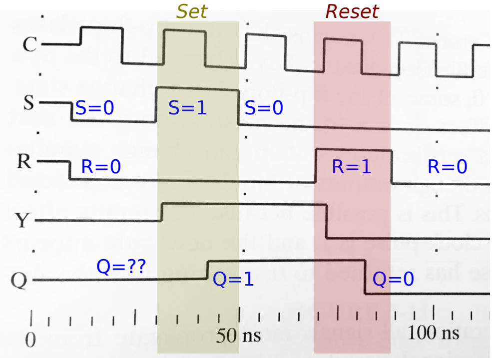
Example: Master Slave SR Flip-Flop and Timing

- Shows how a flip-flop (combination of two latches) stores a bit
- Set to 1: S=1, R=0
- Set to 0: S=0, R=1



State Transition Table

S	R	Q_{next}	Action
0	0	Q	hold state
0	1	0	reset
1	0	1	set
1	1	X	not allowed

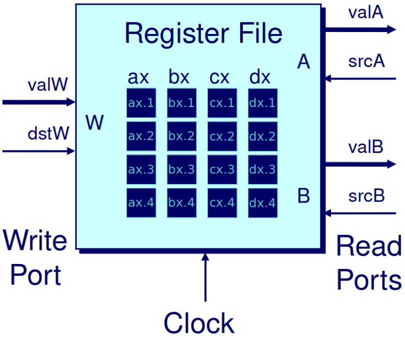


Registers: a form of Static RAM (SRAM)

- Combine 4 flip-flops (each storing one bit) and one has an 4-bit **register**: circuitry that holds a changeable multi-bit quantity
- Combine more flip-flops to get larger registers, 8- 16- 32- 64-bit
- Combine several registers with some access control circuitry (multiplexers) and one has a **register file** containing `%rax %rbx ... %r15`

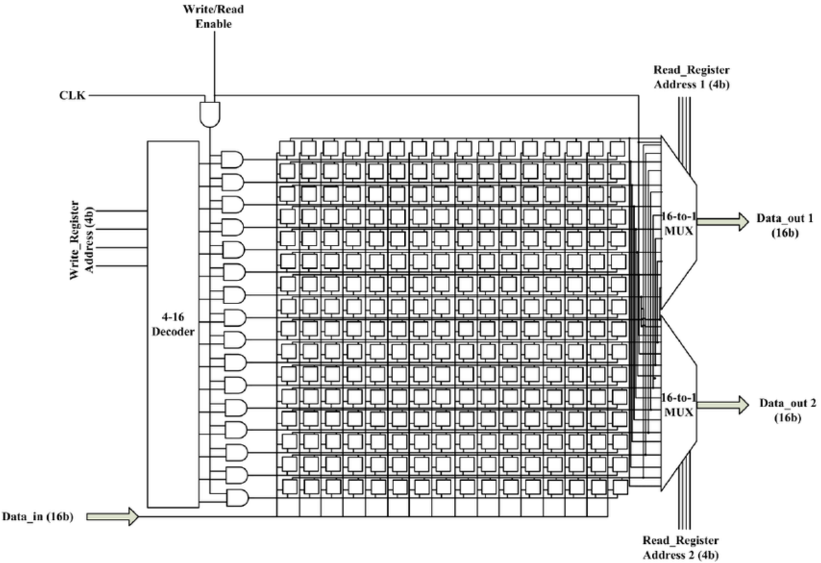
Typical register file allows simultaneous

- read from two regs
- write to one reg



Register File with 4 registers, each with 4 bits

Register File with 16 Regs X 16 Bits + I/O



Source: Mostafa Khatib "Aging Analysis of Datapath Sub-blocks Based on CET Map Model for Negative Bias Temperature Instability (NBTI)", Masters of Science Thesis, Center for Materials and Microsystems, Trento, Italy January 2014

Other Registers/CPU Memory of Note (SRAM)

Instruction Memory/Cache

Fast access to binary opcodes of program text

Program Counter (rip)

Position in instruction memory

Intermediate Results

For internal communication between different parts of the CPU to facilitate pipelining, usually accessible in assembly language

Some Memory Caches

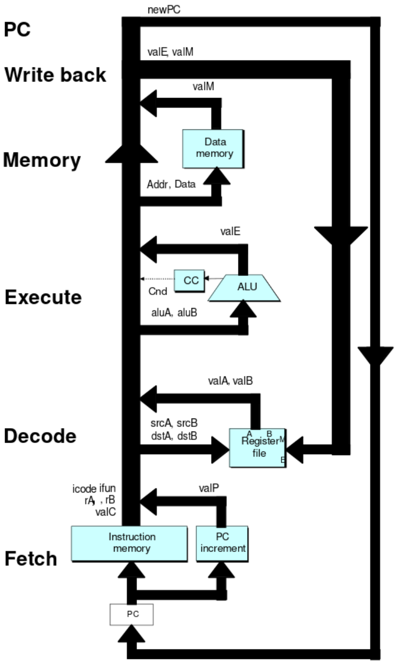
Small, fast cache of main memory close to the cpu has similar circuitry to register file

NOT Main Memory

- While fast, **SRAM is expensive** in terms of transistors/space
- DRAM (dynamic RAM) is slower but compact and cheap enough to scale to gigabytes (will discuss DRAM soon)

The Full Shebang

- Connect an Clock, ALU, and Register file, and you've got a quasi-computer
- Add some instruction decoding, a place to store instructions, and perhaps some main memory and a full computer is born
- Must specify exact encoding of instructions so that signals between gates/units are routed correctly
- Note that processor design to the right is broken into **stages** to help understanding



Exercise: Timing Problems

- Each gate creates a delay: time before output to stabilizes based on new inputs
- Inputs are "allowed" to change on the clock signal's rising edge
- Simplest **sequential** implementation sets clock frequency slow enough for outputs to stabilize each cycle (tick)
- Easy to do, but... it's **slow**

Increasing Efficiency

Propose **two ways** that a complex, multi-part process can be completed faster

- Draw from experience/knowledge
- Think manufacturing, group projects, car wash, Chipotle...

Answer: Timing Problems

General solutions to process speed are familiar to all of us

Assembly Line

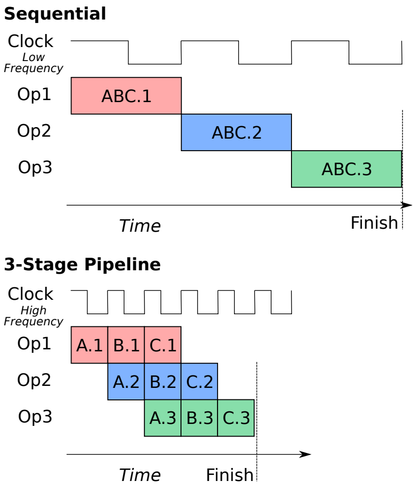
- Break single instruction into multiple “stages” which must all complete
- **Pipelined** processors execute stages simultaneously

Multiple Resources

- Implement multiple functional units and do instructions in parallel
- **Superscalar** processors (and parallel processors)

Pipelining for Efficiency

- Break up processor into "stages" which feed into each other
- Individual instructions like `addl %ecx, %eax` go through each stage
- Instruction completes (*retires*) when all stages complete
- **Begin next instruction when previous clears first stage**
- Some multi-cycle operations like **multiplication** may be pipelined as well



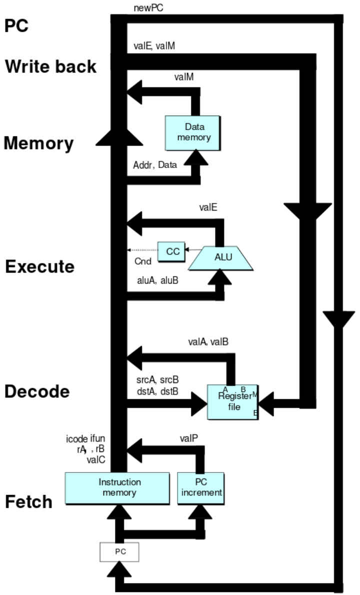
Y86-64: Textbook Processor SEQ vs PIPE

- Textbook discusses 5-stages of a simple CPU design
  1. Fetch next PC
  2. Decode instruction
  3. Execute instruction
  4. Main Memory operations
  5. Write-back to register file
- Diagrams and Hardware Description Language for
  - SEQ: sequential implementation
  - PIPE: pipelined version of processor

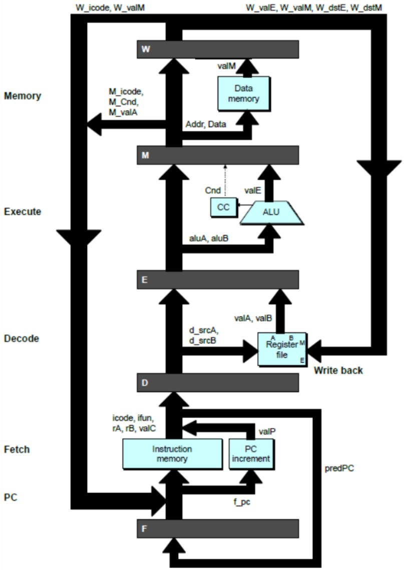
PIPE Version

- Each of 5 stages happens in parallel
- Up to 5 instructions in flight
- Introduces internal registers to facilitate pipeline

Y86-64 SEQ sequential



Y86-64 PIPE 5-stage pipeline



Pipelines Aren't All that and a Bag of Chips

- Pipelining is effective with predictable control flow and independent instructions
- Cases exist in which this doesn't play out: pipeline **hazards**

Data Interdependencies

```
# INDEPENDENT
imull $3, %eax   # mul and add
addl  $1, %edx   # different reg

# DEPENDENT: "Hazard"
imull $3, %eax   # mul and add
addl  $1, %eax   # same reg
```

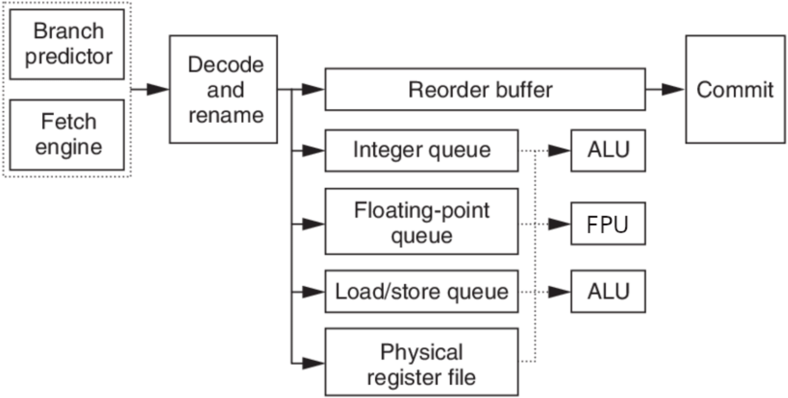
- Dependencies between register results break the pipeline
- Must serialize instructions (sequential execution)

Branching

```
.LOOP:
  addl %edx,%eax
  addl $1,  %ecx
  cmp  %esi,%ecx
  jl   .LOOP # which instruction
  popq %rbx  # next? "hazard"
```

- Modern Processors use **branch prediction** to guess the next instruction
- Incorrect guesses lead to restarting the pipeline

Superscalar Block Diagram



Source: Kilo-Instruction Processors: Overcoming the Memory Wall by Cristal et al.

Note several ALUs, separate queues for different instructions, asynchronous execution of instructions

Superscalar Processing

- Modern processors may have several **functional units** to do arithmetic, logic, other ops
- Allows **instruction-level parallelism**: do two things simultaneously
- Example:

```
# SEQ 1: Multiply only
imull $3, %eax

# SEQ 2: Multiply and Add
imull $3, %eax
addl  $5, %edx
```

- SEQ 1 and SEQ 2 may take the same amount of time
- Separate mult/add units used simultaneously
- Instruction parallelism automatically done at the hardware level leading to naming conventions for processors:
  - "Scalar": sequential only, one thing at a time
  - "**Superscalar**": automatic instruction parallelism, no explicit control
  - "Parallel": explicit instructions that do multiple things simultaneously
- Modern processors are an amalgam of the above

Modern Processors are Weird

Assembly Code as an Interface

- Assembly/Binary Opcodes are a target for high level languages
- Modern processors execute these, guarantee correctness BUT make no guarantees about **how** or in what order
- Most use **very deep** pipelines which must be “fed” to keep speed high
- Has led to exotic processor designs with speculative and out of order execution: keep things in the pipeline
- This hasn’t always gone well: Meltdown / Spectre

Lab10 + HW10: Timing Arithmetic Codes

- Leads to surprising results
- Explainable by considering CPU is pipelined and superscalar
- Timing results vary with different CPUs

Additional Resources the Architecture-Inclined

Building an 8-bit breadboard computer! by Ben Eater (Youtube)

- Discusses many components we briefly touched on in more detail with a very practical bent of using them
- Results in a full CPU + Memory system that you can "see"
- A great introduction to components, breadboards, and general small electronics work

MIT 6.004 Computation Structures, Spring 2017 (Youtube)

- Much deeper detail on many aspects of CPU design
- Includes discussion of Multiplier circuits, power considerations, etc.