CSC 252: Processor Architecture

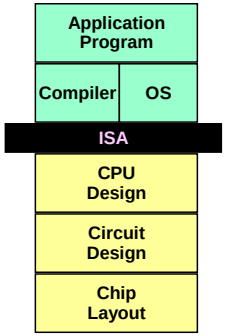
Guest Lecture

by

Sandhya Dwarkadas

Instruction Set Architecture

- Assembly Language View
  - Processor state
    - Registers, memory, …
  - Instructions
    - `addl`, `movl`, `leal`, …
    - How instructions are encoded as bytes
- Layer of Abstraction
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use variety of tricks to make it run fast
    - E.g., execute multiple instructions simultaneously



Basic Issues in Instruction Set Design

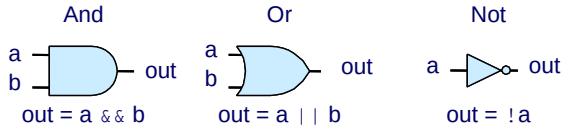
Goal: find a language that makes it easy to build both the hardware and the compiler while maximizing performance and minimizing cost

- What operations (and how many) should be provided
- How (and how many) operands are specified
- What data types and sizes
- How to encode these into consistent instruction formats

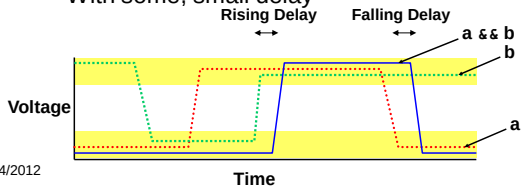
Overview of Logic Design

- Fundamental Hardware Requirements
  - Communication
    - How to get values from one place to another
  - Computation
  - Storage
- Bits are Our Friends
  - Everything expressed in terms of values 0 and 1
  - Communication
    - Low or high voltage on wire
  - Computation
    - Compute Boolean functions
  - Storage
    - Store bits of information

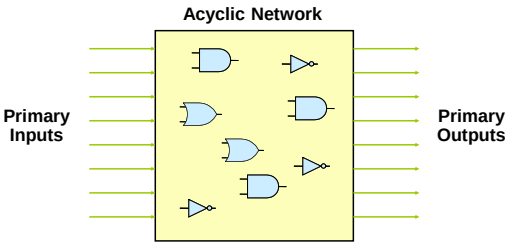
Computing with Logic Gates



- Outputs are Boolean functions of inputs
- Respond continuously to changes in inputs
  - With some, small delay

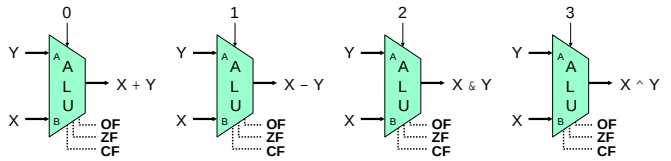


Combinational Circuits



- Acyclic Network of Logic Gates
  - Continously responds to changes on primary inputs
  - Primary outputs become (after some delay) Boolean functions of primary inputs

Arithmetic Logic Unit



- Combinational logic
  - Continuously responding to inputs
- Control signal selects function computed
  - Corresponding to 4 arithmetic/logical operations in Y86
- Also computes values for condition codes

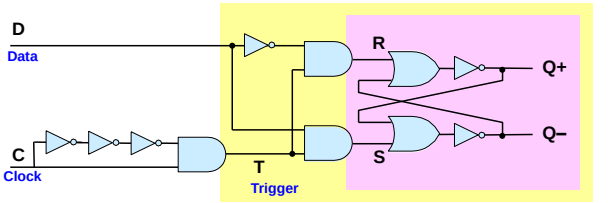
Sequential Logic: Memory and Control

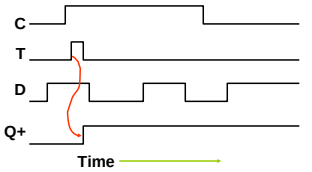
- Sequential:
  - Output depends on the ***current*** input values and the ***previous*** sequence of input values.
  - Are Cyclic:
    - Output of a gate feeds its input at some future time.
  - Memory:
    - Remember results of previous operations
    - Use them as inputs.
  - Example of use:
    - Build registers and memory units.

Clocks

- Signal used to synchronize activity in a processor
- Every operation must be completed in the time between two clock pulses (or rising edges) --- the cycle time
- Maximum clock rate (frequency) determined by the slowest logic path in the circuit (the critical path)

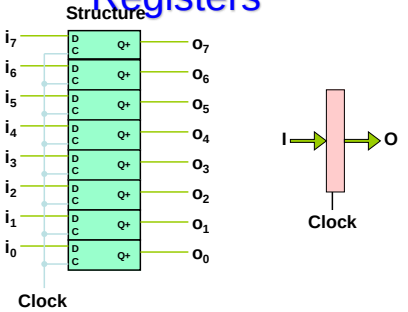
Edge-Triggered Latch





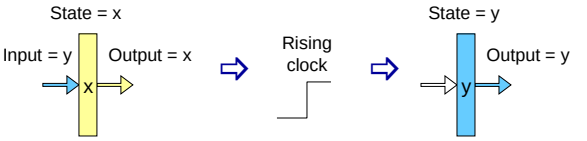
- Only in latching mode for brief period
  - Rising clock edge
- Value latched depends on data as clock rises
- Output remains stable at all other times

Registers



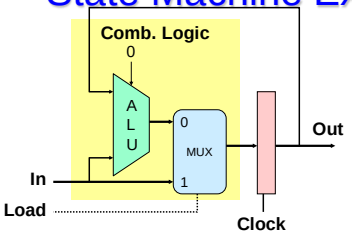
- Stores word of data
  - Different from *program registers* seen in assembly code
- Collection of edge-triggered latches
- Loads input on rising edge of clock

Register Operation

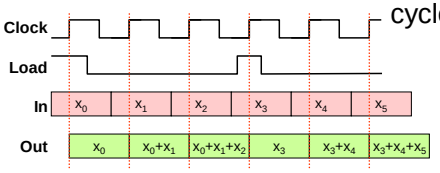


- Stores data bits
- For most of time acts as barrier between input and output
- As clock rises, loads input

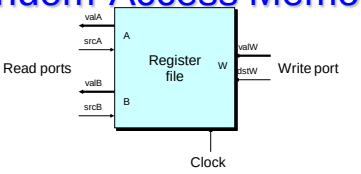
State Machine Example



- Accumulator circuit
- Load or accumulate on each cycle

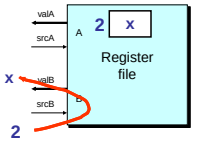


Random-Access Memory

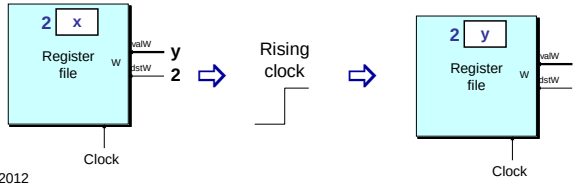


- Stores multiple words of memory
  - Address input specifies which word to read or write
- Register file
  - Holds values of program registers
  - `%eax`, `%esp`, etc.
  - Register identifier serves as address
    - ID 8 implies no read or write performed
- Multiple Ports
  - Can read and/or write multiple words in one cycle
    - Each has separate address and data input/output

Register File Timing

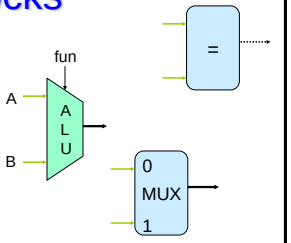


- Reading
  - Like combinational logic
  - Output data generated based on input address
    - After some delay
- Writing
  - Like register
  - Update only as clock rises

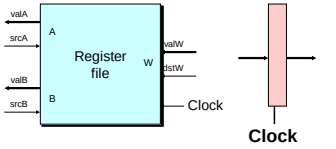


Building Blocks

- Combinational Logic
  - Compute Boolean functions of inputs
  - Continuously respond to input changes
  - Operate on data and implement control

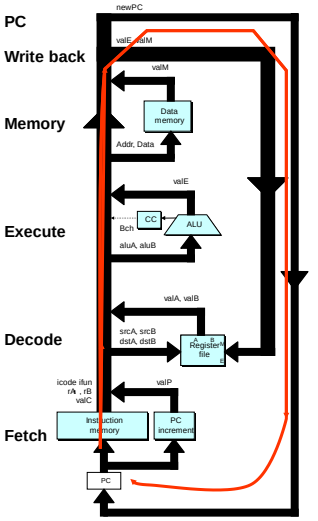


- Storage Elements
  - Store bits
  - Addressable memories
  - Non-addressable registers
  - Loaded only as clock rises



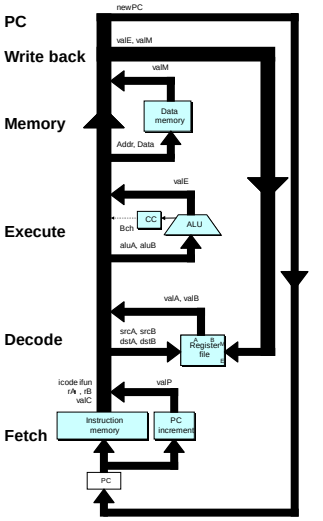
SEQ Hardware Structure

- State
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions
- Instruction Flow
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter

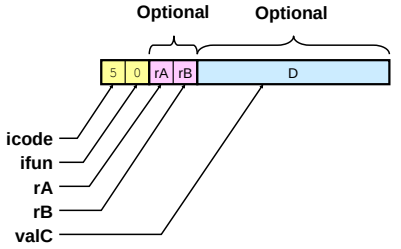


SEQ Stages

- Fetch
  - Read instruction from instruction memory
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers
- PC
  - Update program counter



Instruction Decoding



- Instruction Format
  - Instruction byte — icode:ifun
  - Optional register byte — rA:rB
  - Optional constant word — valC

Executing Arith./Logical Operation

- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - Perform operation
  - Set condition codes
- Memory
  - Do nothing
- Write back
  - Update register
- PC Update
  - Increment PC by 2

Possible Implementation Strategies

- Single-cycle control
- Multi-cycle control
- Pipelined (in-order execution)
- Superscalar and out-of-order execution

Single-Cycle Control

- Cannot use any hardware unit twice in one cycle
  - Multiple datapath elements (e.g., ALU and PC incrementer) if needed for multiple purposes
  - Multiple reads or writes to memory or register file require multiporting
- Clock cycle must accommodate instruction with the longest latency

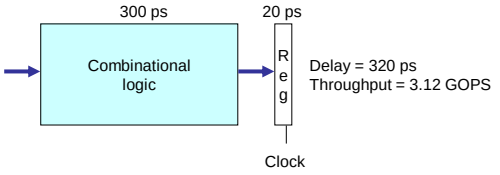
Multi-cycle Control

- Split single instruction into multiple pieces
- Execute individual pieces using hardwired finite state machine (FSM) or microprogramming
- Can finish simple instructions in fewer cycles
- Can run clock faster
- Still execute a single instruction at a time

Pipelined Datapath

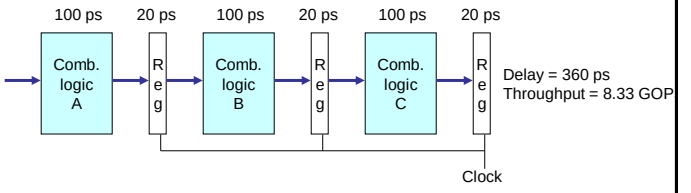
- Multiple instructions overlapped in execution
- Improve throughput, not individual instruction execution time
- Exploit parallelism among instructions in a sequential stream
- Balance length of each stage. Ideally -
  - Time between instrs $_{pipelined}$ = Time between instrs $_{non\text{-}pipelined}$ /Number of pipe stages

Computational Example



- System
  - Computation requires total of 300 picoseconds
  - Additional 20 picoseconds to save result in register
  - Must have clock cycle of at least 320 ps

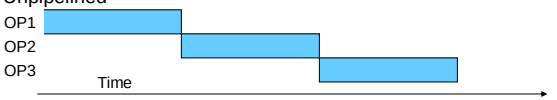
3-Way Pipelined Version



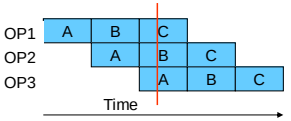
- System
  - Divide combinational logic into 3 blocks of 100 ps each
  - Can begin new operation as soon as previous one passes through stage A.
    - Begin new operation every 120 ps
  - Overall latency increases
    - 360 ps from start to finish

Pipeline Diagrams

- Unpipelined

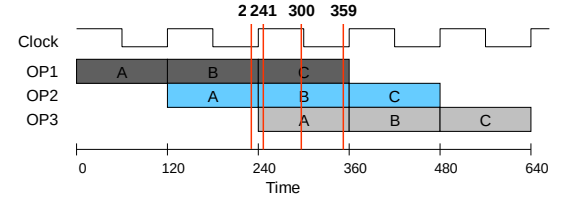


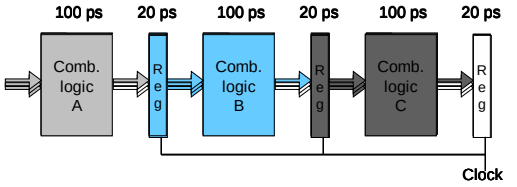
  - Cannot start new operation until previous one completes
- 3-Way Pipelined



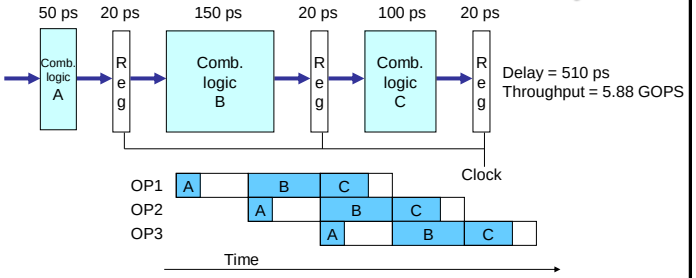
  - Up to 3 operations in process simultaneously

Operating a Pipeline



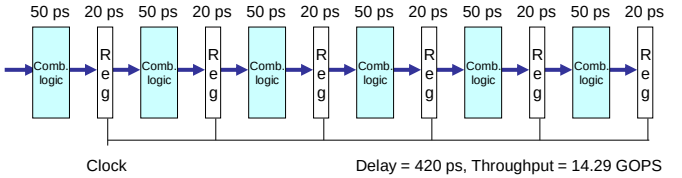


Limitations: Nonuniform Delays



- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Limitations: Register Overhead

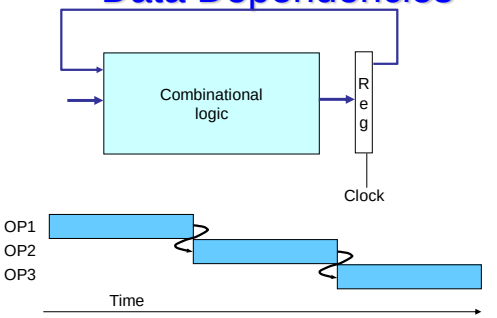


- As we try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

Hazards

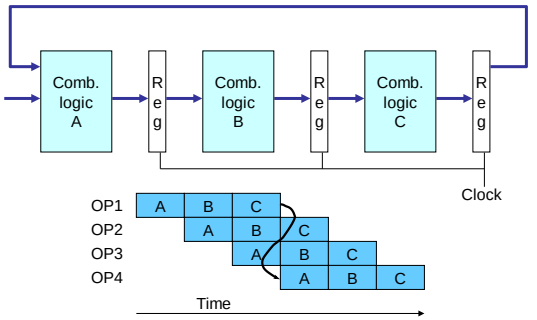
- Structural (e.g., instruction/data fetch)
- Control (e.g., jeq)
- Data (e.g., add followed by sub reading dst register of add)

Data Dependencies



- System
  - Each operation depends on result from preceding one

Data Hazards



– Result does not feed back around in time for next operation
– Pipelining has changed behavior of system

Data Dependencies in Processors

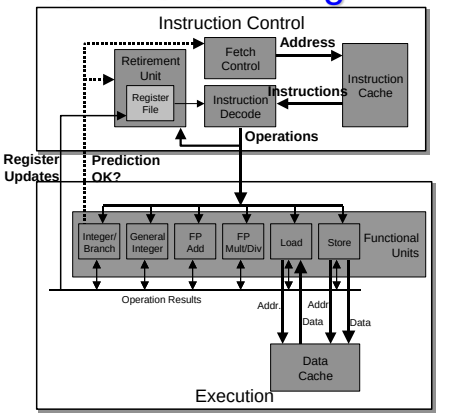
```
1  irmovl $50, %eax
2  addl %eax, %ebx
3  mrmovl 100(%ebx), %edx
```

– Result from one instruction used as operand for another
  • Read-after-write (RAW) dependency
– Very common in actual programs
– Must make sure our pipeline handles these properly
  • Get correct results
  • Minimize performance impact

Pipeline Summary

- Data Hazards
  – Most handled by forwarding
    • No performance penalty
  – Load/use hazard requires one cycle stall
- Control Hazards
  – Cancel instructions when detect mispredicted branch
    • Two clock cycles wasted
  – Stall fetch stage while `ret` passes through pipeline
    • Three clock cycles wasted
- Control Combinations
  – Must analyze carefully
  – Watch for subtle bugs that only arises with unusual instruction combination

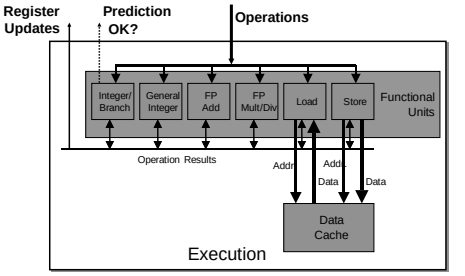
Modern CPU Design



Instruction-level Parallelism

- Pipelining/super-pipelining
- Out-of-order execution
- Super-scalar
  - multiple instructions per pipeline stage
  - Dependences handled in hardware
- Very Large Instruction Word (VLIW)
  - Multiple instructions per pipeline stage
  - Dependences taken care of by compiler

Execution Unit



- Multiple functional units
  - Each can operate independently
- Operations performed as soon as operands available
  - Not necessarily in program order
  - Within limits of functional units
- Control logic
  - Ensures behavior equivalent to sequential program execution

Data Hazards: Forwarding

- The CDC 6600 Score-board architecture
- Tomasulo's generalized forwarding algorithm in the IBM360
  - Used reservation stations and shared buses

Advanced Processor Design Techniques

- Trace caches
- Register renaming – eliminate WAW, WAR hazards
  - Register map table
  - Free list
  - Active list
- Speculative execution
- Value prediction
- Branch prediction
  - Branch history tables for prediction
  - Branch stack to save state prior to branch
  - Branch mask to determine instructions that must be squashed

Processor Summary

- Design Technique
  - Create uniform framework for all instructions
    - Want to share hardware among instructions
  - Connect standard logic blocks with bits of control logic
- Operation
  - State held in memories and clocked registers
  - Computation done by combinational logic
  - Clocking of registers/memories sufficient to control overall behavior
- Enhancing Performance
  - Pipelining increases throughput and improves resource utilization
  - Must make sure maintains ISA behavior

Disclaimer

- Parts of the slides were developed by the course text authors: Dave O'Hallaron and Randy Bryant. The slides are intended for the sole purpose of instruction of computer organization at the University of Rochester. All copyrighted materials belong to their original owner(s).