Lecture 25

- Logistics
  - HW8 due today
  - Ant extra credit due Friday
  - Final exam, Wednesday March 18, 2:30-4:20 pm here
    - Review session Monday, March 16, 4:30 pm, here
- Last lecture
  - Encoding & Partitioning examples
- Today
  - Pipelining & Retiming
  - Control vs Datapath in a simple computer design

Other sequential logic optimization techniques

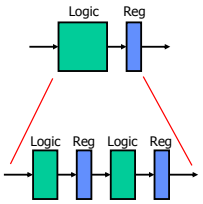
- Pipelining --- allows faster clock speed
- Retiming --- can reduce registers or change delays

Pipelining related definitions

- Latency: Time to perform a computation
  - Data input to data output
- Throughput: Input or output data rate
  - Typically the clock rate
- Combinational delays drive performance
  - Define d ≡ delay through slowest combinational stage
    n ≡ number of stages from input to output
  - Latency ∝ n * d (in sec)
  - Throughput ∝ 1/d (in Hz)

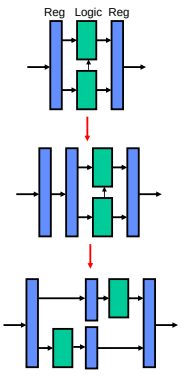
Pipelining

- What?
  - Subdivide combinational logic
  - Add registers between logic
- Why?
  - Trade latency for throughput
  - Increased throughput
    - Reduce logic delays
    - Increase clock speed
  - Increased latency
    - Takes cycles to fill the pipe
  - Increase circuit utilization
    - Simultaneous computations

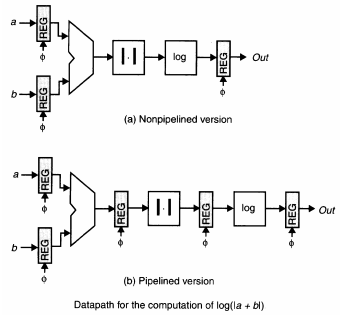


Pipelining

- When?
  - Need throughput more than latency
    - Signal processing
  - Logic delays > setup/hold times
  - Acyclic logic
- Where?
  - At natural breaks in the combinational logic
  - Adding registers makes sense



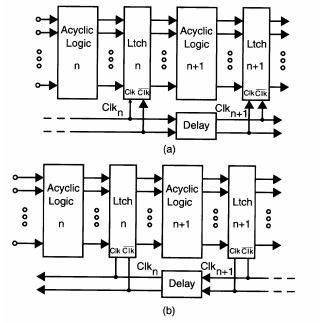
Pipelining example



Datapath for the computation of log(|a + b|)

Pipelining and clock skew

- Which is faster?
- Which is safer?

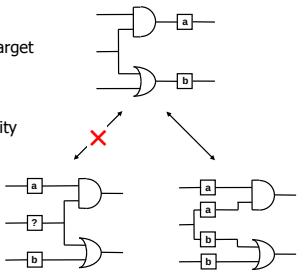


Retiming

- Pipelining adds registers
  - To increase the clock speed
- Retiming moves registers around
  - Reschedules computations to optimize performance
    - Minimize critical path
    - Optimize logic across register boundaries
    - Reduce register count
  - Without altering functionality

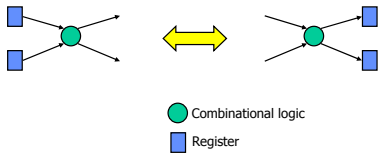
Retiming in a nutshell

- Change position of FFs
  - For speed
  - To suit implementation target
- Retiming modifies state assignment
  - Preserves FSM functionality



Retiming ground rules

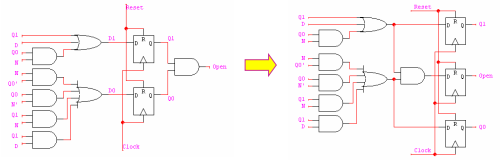
- Rules:
  - Remove one register from each input and add one to each output
  - Remove one register from each output and add one to each input



Retiming examples

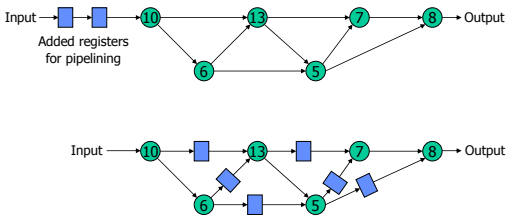
- Reduce register count

- Change output delays



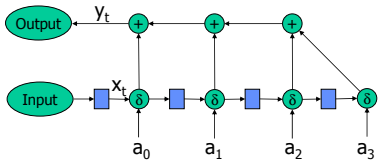
Optimal pipelining

- Add registers
- Use retiming to optimize location



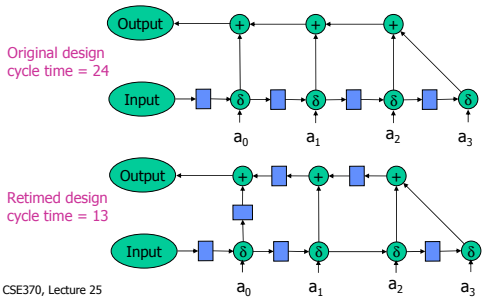
Example: Digital correlator

- $y_t = \delta(x_t, a_0) + \delta(x_{t-1}, a_1) + \delta(x_{t-2}, a_2) + \delta(x_{t-3}, a_3)$
  - δ is a comparator: $\delta(x, a) = 1$ if $x = a$; 0 otherwise
  - y_t is the number of matches between input and pattern $a_0a_1a_2a_3$



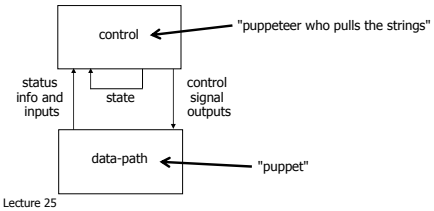
Example: Digital correlator (cont'd)

- Delays: Comparator = 3; adder = 7



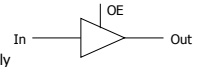
Data-path and control

- Digital hardware systems = data-path + control
  - datapath: registers, counters, combinational functional units (e.g., ALU), communication (e.g., busses)
  - control: FSM generating sequences of control signals that instructs datapath what to do next

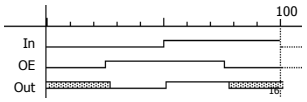


Tri-state gates

- The third value
  - logic values: "0", "1"
  - don't care: "X" (must be 0 or 1 in real circuit!)
  - third value or state: "Z" — high impedance, infinite R, no connection
- Tri-state gates
  - additional input – output enable (OE)
  - output values are 0, 1, and Z
  - when OE is high, the gate functions normally
  - when OE is low, the gate is disconnected from wire at output
  - allows more than one gate to be connected to the same output wire
    - as long as only one has its output enabled at any one time (otherwise, sparks could fly)

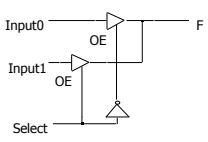


	In	OE	Out
non-inverting tri-state buffer	X	0	Z
	0	1	0
	1	1	1



Tri-state and multiplexing

- When using tri-state logic
  - (1) make sure never more than one "driver" for a wire at any one time
    (pulling high and low at the same time can severely damage circuits)
  - (2) make sure to only use value on wire when its being driven (using a
    floating value may cause failures)
- Using tri-state gates to implement an economical multiplexer



when Select is high
Input1 is connected to F

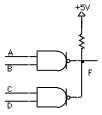
when Select is low
Input0 is connected to F

this is essentially a 2:1 mux

Open-collector gates and wired-AND

- Open collector: another way to connect gate outputs to the same wire
  - gate only has the ability to pull its output low
  - it cannot actively drive the wire high (default – pulled high through resistor)
- Wired-AND can be implemented with open collector logic
  - if A and B are "1", output is actively pulled low
  - if C and D are "1", output is actively pulled low
  - if one gate output is low and the other high, then low wins
  - if both gate outputs are "1", the wire value "floats", pulled high by resistor
    - low to high transition usually slower than it would have been with a gate pulling high
  - hence, the two NAND functions are ANDed together

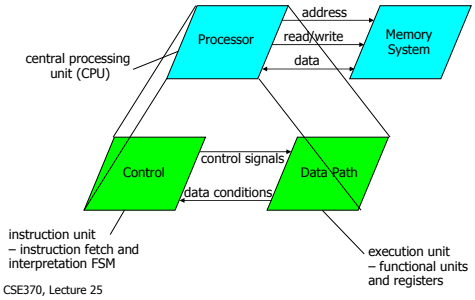
open-collector NAND gates



with ouputs wired together using "wired-AND" to form (AB)'(CD)'

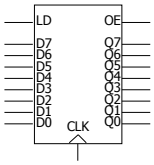
Structure of a computer

- Block diagram view



Registers

- Selectively loaded – EN or LD input
- Output enable – OE input
- Multiple registers – group 4 or 8 in parallel

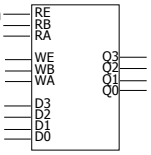


OE asserted causes FF state to be connected to output pins; otherwise they are left unconnected (high impedance)

LD asserted during a lo-to-hi clock transition loads new data into FFs

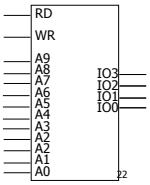
Register files

- Collections of registers in one package
  - two-dimensional array of FFs
  - address used as index to a particular word
  - can have separate read and write addresses so can do both at same time
- 4 by 4 register file
  - 16 D-FFs
  - organized as four words of four bits each
  - write-enable (load)
  - read-enable (output enable)



Memories

- Larger collections of storage elements
  - implemented not as FFs but as much more efficient latches
  - high-density memories use 1 to 5 switches (transitors) per memory bit
- Static RAM – 1024 words each 4 bits wide
  - once written, memory holds forever (not true for denser dynamic RAM)
  - address lines to select word
    - (10 lines for 1024 words)
  - read enable
    - same as output enable
    - often called chip select
    - permits connection of many chips into larger array
  - write enable (same as load enable)
  - bi-directional data lines
    - output when reading, input when writing



Instruction sequencing

- Example – an instruction to add the contents of two registers (Rx and Ry) and place result in a third register (Rz)
- Step 1: get the ADD instruction from memory into an instruction register (IR)
- Step 2: decode instruction
  - instruction in IR has the code of an ADD instruction
  - register indices used to generate output enables for registers Rx and Ry
  - register index used to generate load signal for register Rz
- Step 3: execute instruction
  - enable Rx and Ry output and direct to ALU
  - setup ALU to perform ADD operation
  - direct result to Rz so that it can be loaded into register

Instruction types

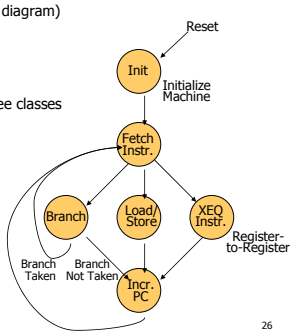
- Data manipulation
  - add, subtract
  - increment, decrement
  - multiply
  - shift, rotate
  - immediate operands
- Data staging
  - load/store data to/from memory
  - register-to-register move
- Control
  - conditional/unconditional branches in program flow
  - subroutine call and return

Elements of the control unit (aka instruction unit)

- Standard FSM elements
  - state register
  - next-state logic
  - output logic (datapath/control signalling)
  - Moore or synchronous Mealy machine to avoid loops unbroken by FF
- Plus additional "control" registers
  - instruction register (IR)
  - program counter (PC)
- Inputs/outputs
  - outputs control elements of data path
  - inputs from data path used to alter flow of program (test if zero)

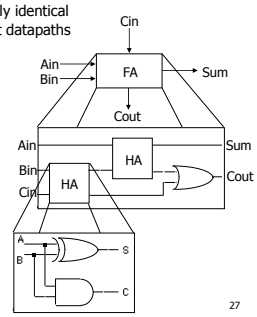
Instruction execution

- Control state diagram (for each diagram)
  - reset
  - fetch instruction
  - decode
  - execute
- Instructions partitioned into three classes
  - branch
  - load/store
  - register-to-register
- Different sequence through diagram for each instruction type



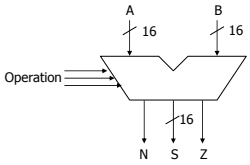
Data path (hierarchy)

- Arithmetic circuits constructed in hierarchical and iterative fashion
  - each bit in datapath is functionally identical
  - 4-bit, 8-bit, 16-bit, 32-bit , 32-bit datapaths



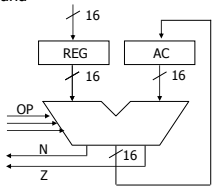
Data path (ALU)

- ALU block diagram
  - input: data and operation to perform
  - output: result of operation and status information



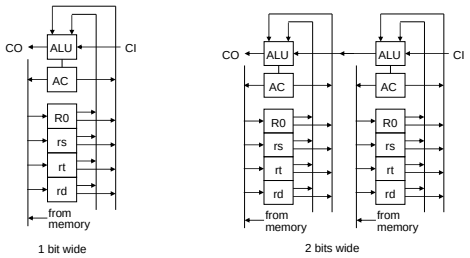
Data path (ALU + registers)

- Accumulator
  - special register
  - one of the inputs to ALU
  - output of ALU stored back in accumulator
- One-address instructions
  - operation and address of one operand
  - other operand and destination is accumulator register
  - AC ← AC op Mem[addr]
  - "single address instructions" (AC implicit operand)
- Multiple registers
  - part of instruction used to choose register operands



Data path (bit-slice)

- Bit-slice concept – iterate to build n-bit wide datapaths



Instruction path

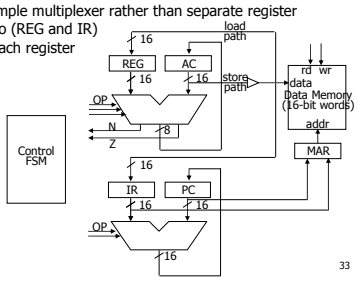
- Program counter
  - keeps track of program execution
  - address of next instruction to read from memory
  - may have auto-increment feature or use ALU
- Instruction register
  - current instruction
  - includes ALU operation and address of operand
  - also holds target of jump instruction
  - immediate operands
- Relationship to data path
  - PC may be incremented through ALU
  - contents of IR may also be required as input to ALU

Data path (memory interface)

- Memory
  - separate data and instruction memory (Harvard architecture)
    - ⇨ two address busses, two data busses
  - single combined memory (Princeton architecture)
    - ⇨ single address bus, single data bus
- Separate memory
  - ALU output goes to data memory input
  - register input from data memory output
  - data memory address from instruction register
  - instruction register from instruction memory output
  - instruction memory address from program counter
- Single memory
  - address from PC or IR
  - memory output to instruction and data registers
  - memory input from ALU output

Block diagram of processor

- Register transfer view of Princeton architecture
  - which register outputs are connected to which register inputs
  - arrows represent data-flow, other are control signals from control FSM
  - MAR may be a simple multiplexer rather than separate register
  - MBR is split in two (REG and IR)
  - load control for each register



Block diagram of processor

- Register transfer view of Harvard architecture
  - which register outputs are connected to which register inputs
  - arrows represent data-flow, other are control signals from control FSM
  - two MARs (PC and IR)
  - two MBRs (REG and IR)
  - load control for each register