Previously

• Boolean Logic
• Gates
• Arithmetic
• Complexity of computations
  – E.g. area and delay for addition

Today

• Sequential Logic
  – Add registers, state
  – Finite-State Machines (FSM)
  – Register Transfer Level (RTL) logic
  – Datapath Reuse
  – Pipelining
  – Latency and Throughput
  – Finite-State Machines with Datapaths (FSMD)

Preclass

• Can we solve the problem entirely using Boolean logic functions?

Latches, Registers

• New element is a state element.
• Canonical instance is a register:
  – remembers the last value it was given until
told to change
  – typically signaled by clock

Why Registers?
Reuse

- In general, we want to reuse our components in time
  - not disposable logic
- How do we do allow guarantee disciplined reuse?

To Reuse Logic...

- Make sure all logic completed evaluation
  - Outputs of gates are valid
    - Meaningful to look at them
  - Gates are "finished" with work and ready to be used again
- Make sure consumers get value
  - Before being overwritten by new calculation (new inputs)

Synchronous Logic Model

- Data starts
  - Inputs to circuit
  - Registers
- Perform combinational (boolean) logic
- Outputs of logic
  - Exit circuit
  - Clocked into registers
- Given long enough clock
  - Think about registers getting values updated by logic on each clock cycle

Issues of Timing...

- …many issues in detailed implementation
  - glitches and hazards in logic
  - timing discipline in clocking
- We’re going to (mostly) work above that level for the most part this term.
  - Will talk about the delay of logic between registers
- Watch for these details in ESE370/570

Preclass

- How do we build an adder for arbitrary input width?

Preclass

- What did the addition of state register(s) do for us?
Added Power

- Process *unbounded* input with finite logic
- State is a *finite* (bounded) representation of what’s happened before
  - finite amount of stuff can remember to synopsized the past
- State allows behavior to depend on past (on context)

Finite-State Machine (FSM)

- Logic core
- Plus registers to hold state

FSM Model

- FSM – a model of computations
- More powerful than Boolean logic functions
- Both
  - Theoretically
  - Practically

Formal FSM Specification

(abstract from implementation)

- An FSM is a sextuple \( M = (K, \Sigma, \delta, s, F, \Sigma_o) \)
  - \( K \) is finite set of states
  - \( \Sigma \) is a finite alphabet for inputs
  - \( s \in K \) is the start state
  - \( F \subseteq K \) is the set of final states
  - \( \Sigma_o \) is a finite set of output symbols
  - \( \delta \) is a transition function from \( K \times \Sigma \) to \( K \times \Sigma_o \)

Finite State Machine

- Less formally:
  - Behavior depends not just on input
    - (as was the case for combinational logic)
  - Also depends on state
  - Can be completely different behavior in each state
  - Logic/output now depends on both
    - state and input

Specifying an FSM

- Logic becomes:
  - if \((\text{state}=s1)\)
    - boolean logic for state 1
      - (including logic for calculate next state)
  - else if \((\text{state}=s2)\)
    - boolean logic for state 2
  - ...
**FSM Specification**

- St1: goto St2
- St2:
  - if (I==0) goto St3
  - else goto St4
- St3:
  - output o0=1
  - goto St1
- St4:
  - output o1=1
  - goto St2

**State Encoding**

- States not (necessarily) externally visible
- We have *freedom* in how to encode them
  - assign bits to states
- Usually want to exploit freedom to minimize implementation costs
  - area, delay, energy
- (there are algorithms to attack – ESE535)

---

**FSM Equivalence**

- Harder than Boolean logic
- Doesn’t have unique canonical form
- Consider:
  - state encoding not change behavior
  - two “equivalent” FSMs may not even have the same number of states
  - can deal with *infinite* (unbounded) input
  - ...so cannot enumerate output in all cases
    - No direct correspondence of a truth table

**FSM Equivalence Flavor**

- Given two FSMs A and B
  - consider the composite FSM AB
  - Inputs wired together
  - Outputs separate
- Ask:
  - is it possible to get into a composite state in which A and B output different symbols?
  - There is a literature on this

**Systematic FSM Design**

- Start with specification
- Can compute Boolean logic for each state
  - if conversion...
    - including next state translation
    - keep state symbolic (s1, s2…)
- Assign state encodings
- Then have combinational logic
  - has current state as part of inputs
  - produces next state as part of outputs
- Design comb. logic and add state registers
Arbitrary Adder

- Work through design as FSM if necessary

RTL

- Register Transfer Level description
- Registers + Boolean logic
- Most likely: what you’ve written in Verilog, VHDL

Datapath Reuse

Reuse: “Waiting” Discipline

- Use registers and timing for orderly progression of data

Example: 4b Ripple Adder

- Recall 2 gates/FA
- How fast can we clock this?
- Min Clock Cycle: 8 gates A, B to S3

Can we do better?

- Clock faster, reuse elements sooner?
Stagger Inputs

• Correct if expecting A,B[3:2] to be staggered one cycle behind A,B[1:0]
• …and succeeding stage expects S[3:2] staggered from S[1:0]

Align Data / Balance Paths

Good discipline to line up pipe stages in diagrams.

Speed

How fast can we clock this?

What is the delay from A,B to S3?

Pipelining and Timing

• Once introduce pipelining
  – Clock cycle = rate of reuse
  – Is not the same as the delay to complete a computation

• Throughput
  – How many results can the circuit produce per unit time
  – If can produce one result per cycle,
    • Reciprocal of clock period
  • Throughput this design?

• Latency
  – How long does it take to produce one result
  – Product of clock cycle and number of clocks between input and output
  • Latency this design?
Example: 4b RA pipe 2

- Recall 2 gates/FA
- Latency and Throughput:
  - Latency: 8 gates to S3
  - Throughput: 1 result / 4 gate delays max

Deeper?

- Can we do it again?
- What's our limit?
- Why would we stop?

More Reuse

- Saw could pipeline and reuse FA more frequently
- Suggests we're wasting the FA part of the time in non-pipelined
  - What is FA3 doing while FA0 is computing?

More Reuse (cont.)

- If we're willing to take 4 gate-delay units, do we need 4 FAs?

Ripple Add (pipe view)

Can pipeline to FA

What if don't need the throughput?

If don't need throughput, reuse FA on SAME addition.

Bit Serial Addition

Assumes LSB first ordering of input data.
Bit Serial Addition: Pipelining

- Latency and throughput?
- Latency: 8 gate delays
  - 10 for 5th output bit
- Throughput: 1 result / 10 gate delays
- Registers do have time overhead
  - setup, hold time, clock jitter

Multiplication

- Can be defined in terms of addition
- Ask you to play with implementations and tradeoffs in homework 2
  - Out today
  - Pickup from syllabus page on web

Compute Function

- Compute: \( y = Ax^2 + Bx + C \)
- Assume
  - \( D(\text{Mpy}) > D(\text{Add}) \)
    - E.g. \( D(\text{Mpy}) = 24, D(\text{Add}) = 8 \)
  - \( A(\text{Mpy}) > A(\text{Add}) \)
    - E.g. \( A(\text{Mpy}) = 64, A(\text{Add}) = 8 \)

Spatial Quadratic

- \( D(\text{Quad}) = 2D(\text{Mpy}) + D(\text{Add}) = 56 \)
- Throughput \( 1/(2D(\text{Mpy}) + D(\text{Add})) = 1/56 \)
- Area \( A(\text{Quad}) = 3A(\text{Mpy}) + 2A(\text{Add}) = 208 \)

Pipelined Spatial Quadratic

- \( D(\text{Quad}) = 3D(\text{Mpy}) = 72 \)
- Throughput \( 1/D(\text{Mpy}) = 1/24 \)
- Area \( A(\text{Quad}) = 3A(\text{Mpy}) + 2A(\text{Add}) + 6A(\text{Reg}) = 232 \)
Quadratic with Single Multiplier and Adder?

• We’ve seen reuse to perform the same operation
  – pipelining
  – bit-serial, homogeneous datapath
• We can also reuse a resource in time to perform a different role.

Repeated Operations

• What operations occur multiple times in this datapath?
  – x*x, A*(x*x), B*x
  – (Bx)+c, (A*x*x)+(Bx+c)

Quadratic Datapath

• Start with one of each operation
  • (alternatives where build multiply from adds…e.g. homework)

Quadratic Datapath

• Multiplier serves multiple roles
  – x*x
  – A*(x*x)
  – B*x
• Will need to be able to steer data (switch interconnections)

Quadratic Datapath

• Multiplier servers multiple roles
  – x*x
  – A*(x*x)
  – B*x
• x, x*x, x
• x,A,B
Quadratic Datapath

• Adder serves multiple roles
  – (Bx)+c
  – (A'*x'*x)+(Bx+c)
• one always mpy output
• C, Bx+C

Quadratic Control

• Now, we just need to control the datapath
  • What control?
  • Control:
    – LD x
    – LD x'*x
    – MA Select
    – MB Select
    – AB Select
    – LD Bx+C
    – LD Y

FSMD

• FSMD = FSM + Datapath
• Stylization for building controlled datapaths such as this (a pattern)
• Of course, an FSMD is just an FSM
  – it’s often easier to think about as a datapath
  – synthesis, place and route tools have been notoriously bad about discovering/exploiting datapath structure
Quadratic FSMD Control

- S0: if (go) LD_X; goto S1
  - else goto S0
- S1: MA_SEL=x, MB_SEL[1:0]=x, LD_x*x
  - goto S2
- S2: MA_SEL=x, MB_SEL[1:0]=B
  - goto S3
- S3: AB_SEL=C, MA_SEL=x*x, MB_SEL=A
  - goto S4
- S4: AB_SEL=Bx+C, LD_Y
  - goto S0

Quadratic FSM

- D(mux3)=D(mux2)=1
- A(mux2)=2
- A(mux3)=3
- A(QFSM) ~= 10
- Latency/Throughput/Area?
  - Latency: 5*(D(MPY)+D(mux3)) = 125
  - Throughput: 1/Latency = 1/125
  - Area: A(Mpy)+A(Add)+5^A(Reg) +2^A (Mux2)+A(Mux3)+A(QFSM) = 109

Admin: Reminder

- Chrome and Blackboard don’t mix
- Next homework due Monday
- Office hours W2pm
  - 30 minutes after class

Big Ideas
[MSB Ideas]

- Registers allow us to reuse logic
- Can implement any FSM with gates and registers
- Pipelining
  - increases parallelism
  - allows reuse in time (same function)
- Control and Sequencing
  - reuse in time for different functions
- Can tradeoff Area and Time

Big Ideas
[MSB-1 Ideas]

- RTL specification
- FSMD idiom