CS152: Computer Systems Architecture
Circuits Recap

Sang-Woo Jun
Winter 2019

Large amount of material adapted from MIT 6.004, “Computation Structures”, Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”, and CS 152 Slides by Isaac Scherson
Course outline

- Part 1: The Hardware-Software Interface
  - What makes a ‘good’ processor?
  - Assembly programming and conventions

- Part 2: Recap of digital design
  - Combinational and sequential circuits
  - How their restrictions influence processor design

- Part 3: Computer Architecture
  - Computer Arithmetic
  - Simple and pipelined processors
  - Caches and the memory hierarchy

- Part 4: Computer Systems
  - Operating systems, Virtual memory
The digital abstraction

“Building Digital Systems in an Analog World”
The digital abstraction

- Electrical signals in the real world is analog
  - Continuous signals in terms of voltage, current,

- Modern computers represent and process information using discrete representations
  - Typically binary (bits)
  - Encoded using ranges of physical quantities (typically voltage)
Aside: Historical analog computers

- Computers based on analog principles have existed
  - Uses analog characteristics of capacitors, inductors, resistors, etc to model complex mathematical formulas
    - Very fast differential equation solutions!
    - Example: Solving circuit simulation would be very easy if we had the circuit and were measuring it

- Some modern resurgence as well!
  - Research on sub-modules performing fast non-linear computation using analog circuitry

Why are digital systems desirable?

Hint: Noise
Using voltage digitally

- **Key idea**
  - Encode two symbols, “0” and “1” (1 bit) in an analog space
  - And use the same convention for every component and wire in system

![Diagram of voltage levels and fuzzy area]

- **Attempt #1:**
  - V < V_{TH} interpreted as “0”
  - V \geq V_{TH} interpreted as “1”

- **Attempt #2:**
  - V \leq V_L interpreted as “0”
  - V_L < V < V_H interpreted as “Undefined”
  - V_H \geq V interpreted as “1”

- Fuzzy area!

V_L and V_H are enforced during component design and manufacture

Source: MIT 6.004 2019 L05
Handling noise

- When a signal travels between two modules, there will be noise
  - Temperature, electromagnetic fields, interaction with surrounding modules, ...

- What if $V_{\text{out}}$ is barely lower than $V_L$, or barely higher than $V_H$?
  - Noise may push the signal into invalid range
  - Rest of the system runs into undefined state!

- Solution: Output signals use a stricter range than input
Voltage Transfer Characteristic

- Example component: Buffer
  - A simple digital device that copies its input value to its output

- Voltage Transfer Characteristic (VTC):
  - Plot of $V_{\text{out}}$ vs. $V_{\text{in}}$ where each measurement is taken after any transients have died out.
  - Not a measure of circuit speed!
    - Only determines behavior under static input

- Each component generates a new, “clean” signal!
  - Noise from previous component corrected

Source: MIT 6.004 2019 L05
Benefits of digital systems

- Digital components are “restorative”
  - Noise is cancelled at each digital component
  - Very complex designs can be constructed on the abstraction of digital behavior

- Compare to analog components
  - Noise is accumulated at each component
  - Lay example: Analog television signals! (Before 2000s)
    - Limitation in range, resolution due to transmission noise and noise accumulation
    - Contrary: digital signals use repeaters and buffers to maintain clean signals

Source: “Does TV static have anything to do with the Big Bang?” How it works, 2012
Digital circuit design recap
Combinational and sequential circuits

- Two types of digital circuits
- Combinational circuit
  - Output is a function of current input values
    - output = f(input)
    - Output depends exclusively on input
- Sequential circuit
  - Have memory ("state")
    - Output depends on the "sequence" of past inputs
What constitutes combinational circuits

1. Input
2. Output
3. Functional specifications
   - The value of the output depending on the input
   - Defined in many ways!
   - Boolean logic, truth tables, hardware description languages, We’ve done this in CS151

4. Timing specifications
   - Given dynamic input, how does the output change over time?
Timing specifications of combinational circuits

- Propagation delay ($t_{PD}$)
  - An upper bound on the delay from valid inputs to valid outputs
  - Restricts how fast input can be consumed
    (Too fast input $\rightarrow$ output cannot change in time, or undefined output)

A good circuit has low $t_{PD}$
- Faster input
- Higher performance

How do we get low $t_{PD}$?
Timing specifications of combinational circuits

- Propagation delay ($t_{CD}$)
  - A lower bound on the delay between input change to output change
  - Guarantees that output will not change within this timeframe regardless of what happens to input

Example: Inverter

No promises during
The basic building block: CMOS transistors ("Complementary Metal–Oxide–Semiconductor")

Everything is built as a network of transistors!

Source: MIT 6.004 2019 L09
The basic building block: CMOS FETs

- Remember CS151 – FETs come in two varieties, and are composed to create Boolean logic.
Making chips out of transistors…?
The basic building block 2: Standard cell library

- **Standard cell**
  - Group of transistor and interconnect structures that provides a boolean logic function
    - Inverter, buffer, AND, OR, XOR, ...
  - For a specific implementation technology/vendor/etc...
  - Also includes physical characteristic information

- **Eventually, chips designs are expressed as a group of standard cells networked via wires**
  - Among what is sent to a fab plant

**Gate** | **Delay (ps)** | **Area (μ²)**
---|---|---
Inverter | 20 | 10
Buffer | 40 | 20
AND2 | 50 | 25
NAND2 | 30 | 15
OR2 | 55 | 26
NOR2 | 35 | 16
AND4 | 90 | 40
NAND4 | 70 | 30
OR4 | 100 | 42
NOR4 | 80 | 32

Example:

Various components have different delays and area! The actual numbers are not important right now.
Back to propagation delay of combinational circuits

- A chain of logic components has additive delay
  - The “depth” of combinational circuits is important
- The “critical path” defines the overall propagation delay of a circuit

Example: A full adder

Critical path of three components
\[ t_{PD} = t_{PD}(\text{xor2}) + t_{PD}(\text{and2}) + t_{PD}(\text{or2}) \]

Source: en:User:Cburnett @ Wikimedia
Sequential circuits

- Combinational circuits on their own are not very useful
- Sequential logic has memory ("state")
  - State acts as input to internal combinational circuit
  - Subset of the combinational circuit output updates state
Synchronous sequential circuits

- “Synchronous”: all operations are aligned to a shared clock signal
  - Speed of the circuit determined by the delay of its longest critical path
  - For correct operation, all paths must be shorter than clock speed
  - Either simplify logic, or reduce clock speed!
Timing behavior of state elements

- Synchronous state elements also add timing complexities
  - Beyond propagation delay and contamination delay
- Propagation delay ($t_{PD}$) of state elements
  - Rising edge of the clock to valid output from state element
- Setup time ($t_{SETUP}$)
  - State element should have held correct data for $t_{SETUP}$ before clock edge
- Hold time ($t_{HOLD}$)
  - State element should hold correct data for $t_{HOLD}$ after clock edge
- Contamination delay ($t_{CD}$)
  - State element output should not change for $t_{CD}$ after clock change
Timing behavior of state elements

Meeting the setup time constraint

- “Processing must fit in clock cycle”
- After rising clock edge,
- \( t_{PD}(\text{State element } 1) + t_{PD}(\text{Combinational logic}) + t_{\text{SETUP}}(\text{State element } 2) \)
- must be smaller than the clock period

Otherwise, “timing violation”
Timing behavior of state elements

- Meeting the **hold time** constraint
  - “Processing should not effect state too early”
  - After rising clock edge,
  - $t_{CD}(\text{State element 1}) + t_{CD}(\text{Combinational logic})$ must be larger than $t_{HOLD}(\text{State element 2})$
  - Guaranteed time output will not change
Real-world implications

- Constraints are met via Computer-Aided Design (CAD) tools
  - Cannot do by hand!
  - Given a high-level representation of function, CAD tools will try to create a physical circuit representation that meets all constraints

- Rule of thumb: Meeting **hold time** is typically not difficult
  - e.g., Adding a bunch of buffers can add enough $t_{CD}$ (Sequential Circuit)

- Rule of thumb: Meeting **setup time** is often difficult
  - Somehow construct shorter critical paths, or
  - reduce clock speed (We want to avoid this!)

How do we create shorter critical paths for the same function?
Simplified introduction to placement/routing

- Mapping state elements and combinational circuits to limited chip space
  - Also done via CAD tools
  - May add significant propagation delay to combinational circuits

- Example:
  - Complex combinational circuits 1 and 2 accessing state A
  - Spatial constraints push combinational circuit 4 far from state A
  - Path from B to A via 4 is now very long!

- Rule of thumb:
  - One comb. should not access too many state
  - One state should not be used by too many comb.
Looking back:
Why are register files small?

- Why are register files 32-element? Why not 1024 or more?

Hierarchical design of an 8x1 multiplexer:

Propagation delay increases with more registers!
Real-world example

- Back in 2002 (When frequency scaling was going strong)
  - Very high frequency (multi-GHz) meant:
  - ... setup time constraint could tolerate
  - ... up to 8 inverters in its critical path
  - Such stringent restrictions!

Can we even fit a 32-bit adder there? No!
Eight great ideas

- Design for Moore’s Law
- Use abstraction to simplify design
- Make the common case fast
- Performance via parallelism
- Performance via pipelining
- Performance via prediction
- Hierarchy of memories
- Dependability via redundancy

But before we start...
Segue: High-Level Hardware-Description Language

- Modern circuit design is aided heavily by Hardware-Description Languages
  - Relatively high-level description to compiler
  - Toolchain performs “synthesis”, translating them into gates, also place, route, etc
  - High-end chips require human intervention in each stage for optimization

- Wide spectrum of languages and tools
  - Register-Transfer-Level (RTL) languages: Verilog, VHDL, ...
    - Efficient, difficult to program
    - Registers (state), and combinational logic
  - “High-Level Synthesis”: Uses familiar software programming languages
    - Easy to program, inefficient
    - C-to-gates, OpenCL, ...
Bluespec System Verilog (BSV)

- “High-level HDL without performance compromise”
- Comprehensive type system and type-checking
  - Types, enums, structs
- Static elaboration, parameterization (Kind of like C++ templates)
  - Efficient code re-use
- Efficient functional simulator (bluesim)  
  printf’s and user input during simulation!
- Most expertise transferrable between Verilog/Bluespec

In a comparison with a 1.5 million gate ASIC coded in Verilog, Bluespec demonstrated a 13x reduction in source code, a 66% reduction in verification bugs, equivalent speed/area performance, and additional design space exploration within time budgets.

-- PineStream consulting group
Bluespec System Verilog (BSV) High-Level

- Everything organized into “Modules”
  - Modules have an “interface” which other modules use to access state
  - A Bluespec model is a single top-level module consisting of other modules, etc

- Modules consist of state (other modules) and behavior
  - State: Registers, FIFOs, RAM, ...
  - Behavior: Rules, Interface
Greatest Common Divisor Example

- Euclid’s algorithm for computing the greatest common divisor (GCD)

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>subtract</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>subtract</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>swap</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>subtract</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>subtract</td>
</tr>
</tbody>
</table>

answer
module mkGCD (GDCIfc);
  Reg#(Bit#(32)) x <- mkReg(0);
  Reg#(Bit#(32)) y <- mkReg(0);
  FIFOF#(Bit#(32)) outQ <- mkSizedFIFOF(2);

rule step1 ((x > y) && (y != 0));
  x <= y; y <= x;
endrule
rule step2 (( x <= y) && (y != 0));
  y <= y-x;
  if ( y-x == 0 ) begin
    outQ.enq(x);
  end
endrule

method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
  x <= a; y <= b;
endmethod
method ActionValue#(Bit#(32)) result();
  outQ.deq;
  return outQ.first;
endmethod
endmodule

Sub-modules
Module “mkReg” with interface “Reg”,
type parameter Int#(32),
module parameter “0”*

*mkReg implementation sets initial value to “0”

outQ has a module parameter “2”*

*mkSizedFIFOF implementation sets FIFO size to 2
module mkGCD (GDCIfc);
  Reg#(Bit#(32)) x <= mkReg(0);
  Reg#(Bit#(32)) y <= mkReg(0);
  FIFOF#(Bit#(32)) outQ <= mkSizedFIFOF(2);
rule step1 ((x > y) && (y != 0));
   x <= y; y <= x;
endrule
rule step2 ((x <= y) && (y != 0));
   y <= y-x;
   if (y-x == 0) begin
      outQ.enq(x);
   end endrule
method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
   x <= a; y <= b;
endmethod
method ActionValue#(Bit#(32)) result();
   outQ.deq;
   return outQ.first;
endmethod endmodule

Rules are atomic transactions
“fire” whenever condition (“guard”) is met
module mkGCD (GDCIfc);
    Reg#(Bit#(32)) x <- mkReg(0);
    Reg#(Bit#(32)) y <- mkReg(0);
    FIFOF#(Bit#(32)) outQ <- mkSizedFIFOF(2);

    rule step1 ((x > y) && (y != 0));
        x <= y; y <= x;
    endrule

    rule step2 (( x <= y) && (y != 0));
        y <= y - x;
        if ( y-x == 0 ) begin
            outQ.enq(x);
        end
    endrule

    method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
        x <= a; y <= b;
    endmethod

    method ActionValue#(Bit#(32)) result();
        outQ.deq;
        return outQ.first;
    endmethod
endmodule

---

**State**
- `module mkGCD (GDCIfc);`
- `Reg#(Bit#(32)) x <- mkReg(0);`
- `Reg#(Bit#(32)) y <- mkReg(0);`
- `FIFOF#(Bit#(32)) outQ <- mkSizedFIFOF(2);`

**Rules (Behavior)**
- `rule step1 ((x > y) && (y != 0));`
  - `x <= y; y <= x;`
  
- `rule step2 (( x <= y) && (y != 0));`
  - `y <= y - x;`
  - `if ( y-x == 0 ) begin`
    - `outQ.enq(x);`
  - `end`

**Interface (Behavior)**
- `method Action start(Bit#(32) a, Bit#(32) b) if (y==0);`
  - `x <= a; y <= b;`
- `endmethod`
- `method ActionValue#(Bit#(32)) result();`
  - `outQ.deq;`
  - `return outQ.first;`
- `endmethod`

---

Interface methods are also atomic transactions
Can be called only when guard is satisfied
When guard is not satisfied, rules that call it cannot fire
Bluespec Modules – Interface

- Modules encapsulates state and behavior (think C++/Java classes)
- Can be interacted from the outside using its “interface”
  - Interface definition is separate from module definition
  - Many module definitions can share the same interface: Interchangeable implementations
- Interfaces can be parameterized
  - Like C++ templates “FIFO#(Bit#(32))”
  - Not important right now

```plaintext
interface GDCIfc;
  method Action start(Bit#(32) a, Bit#(32) b);
  method ActionValue#(Bit#(32)) result();
endinterface

module mkGCD (GDCIfc);
  ...
  method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
    x <= a; y <= b;
  endmethod
  method ActionValue#(Bit#(32)) result();
    outQ.deq;
    return outQ.first;
  endmethod
endmodule
```
Bluespec Module – Interface Methods

- Three types of methods
  - Action: Takes input, modifies state
  - Value: Returns value, does not modify state
  - ActionValue: Returns value, modifies state

- Methods can have “guards”
  - Does not allow execution unless guard is True
Sequential circuits in Bluespec

- A Bluespec rule represents a state transfer via combinational circuits
  - Much like Verilog “always” and VHDL “process”
  - Can call methods of other modules
    - e.g., `outQ.enq` – Introduces implicit guard if `outQ` is empty

```plaintext
rule step2 ((x <= y) && (y != 0));
  y <= y-x;
  if ( y-x == 0 ) begin
    outQ.enq(x);
  end
endrule
```
Bluespec Rules Are Atomic Transactions

- Each statement in rule only has access to state values from before rule began firing
- Each statement executes independently, and state update happens once as the result of rule firing
  - e.g.,
    // x == 0, y == 1
    x <= y; y <= x; // x == 1, y == 0
  - e.g.,
    // x == 0, y == 1
    x <= 1; x <= y; // write conflict error!

```plaintext
rule step2 ((x <= y) && (y != 0));
    y <= y-x;
    if ( y-x == 0 ) begin
        outQ.enq(x);
    end
endrule
```

Fires if:
1. \(x <= y \&\& y != 0 \&\& y-x == 0 \&\& outQ.notFull\)
   or
2. \(x <= y \&\& y != 0 \&\& y-x != 0\)
Bluespec State – FIFO

- One of the most important modules in Bluespec
- Fixed size queue!
- Parameterized interface with guarded methods
  - e.g., testQ.enq(data); // Action method. Blocks when full
  - testQ.deq; // Action method. Blocks when empty
  - dataType d = testQ.first; // Value method. Blocks when empty
- Provided as library
  - Needs “import FIFO::*;” at top

```
FIFO#(Bit#(32)) testQ <- mkSizedFIFO(2);
rule enqdata; // whole rule does not fire if testQ is full
  if ( x ) y <= z;
  testQ.enq(32’h0);
endrule
```
Bluespec rules:
State and temporary variables

- **State**: Defined outside rules, data stored across clock cycles
  - All state updates happen atomically
  - `Reg#(...)`, `FIFO#(...)`
  - Register state assignment uses “<=“

- **Temporary variables**: Defined within rules, data local to a rule execution
  - Follows sequential semantics similar to software languages
  - Temporary variable value assignment uses “=“
Bluespec rules:
State and temporary variables

Temporary variables behave as you would expect

```plaintext
Reg#(Bit#(32)) a <- mkReg(1); // State
Reg#(Bit#(32)) b <- mkReg(4); // State
rule rule_a;
  Bit#(32) c = a+1; // Temporary variable c == 2
  Bit#(32) d = (c + b)/2; // Temporary variable d == 3
  a <= d; // State a == 3 after this cycle
  b <= a+d; // State b == 4 after this cycle
endrule
```
Peek into a RISC-V processor in Bluespec

Processor.bsv

```hs
interface ProcessorIfc;
    method ActionValue#(MemReq32) memReq;
    method Action memResp(Word data);
endinterface

module mkProcessor(ProcessorIfc);
    Reg#(Word) pc <- mkReg(0);
    RFile2R1W rf <- mkRFile2R1W;
    MemorySystemIfc mem <- mkMemorySystem;
    rule doFetch (stage == Fetch);
        let next_pc = pc + 4;
endrule

::
```

Top.bsv

```hs
module mkTop(Empty);
    ProcessorIfc proc <- mkProcessor;
    ::
```

Be mindful of propagation delays of rules!

- It’s easy to lose sense of delays when working with high-level languages
  - What does my code translate to?
- Many arithmetic primitives provided as a vendor-supplied library
  - Addition, multiplication, etc, actually consist of complex circuits

```
let a = b & c;
```

One AND gate, corresponding delay

```
let a = b<<1;
```

Simple re-labeling of wires, no delay

```
let a = b<<s;
```

but

Instantiates barrel shifter!
Very large delay
(Especially if s has many bits)
Suddenly not meeting timing!
Some other common pitfalls

- **Modulo (%)**
  - if ( (counter +1 ) % 32 ) begin ...
  - Replace with:
    - if (counter + 1 == 32 ) begin
      counter <= 0; ...
  - If power of two, use bit masks: if (counter & 32’b11111 == 32’b11111 ) begin ...

- **Variable index assignment**
  - array_arg[idx] <= x; //Where idx is a dynamic value (Reg#)
  - Replace with: Pipelined scatter module. (High latency, low \( t_{PD} \))
    - Will be introduced later

- **Div/Mult**
  - If operand is always power of two, use shifts! (5 bit operand to shift 32 bits)
Back to pipelining
Performance Measures

- Two metrics when designing a system
  1. Latency: The delay from when an input enters the system until its associated output is produced
  2. Throughput: The rate at which inputs or outputs are processed

- The metric to prioritize depends on the application
  - Embedded system for airbag deployment? **Latency**
  - General-purpose processor? **Throughput**
Performance of Combinational Circuits

- For combinational logic
  - latency = \( t_{PD} \)
  - throughput = \( 1/t_{PD} \)

Reg#(...) \( x \leftarrow \text{mkReg}(...) \);
Reg#(...) \( y \leftarrow \text{mkReg}(...) \);
rule \( p \);
  let \( fv = F(x) \);
  let \( gv = G(x) \);
  let \( pv = H(fv, gv) \);
  \( y \leftarrow pv \);
endrule

Is this an efficient way of using hardware?

Source: MIT 6.004 2019 L12
Pipelined Circuits

- Pipelining by adding registers to hold F and G’s output
  - Now F & G can be working on input $X_{i+1}$ while H is performing computation on $X_i$
  - A 2-stage pipeline!
  - For input X during clock cycle j, corresponding output is emitted during clock j+2.

Assuming latencies of 15, 20, 25...

Source: MIT 6.004 2019 L12
Pipelined Circuits

<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unpipelined</td>
<td>45</td>
<td>1/45</td>
</tr>
<tr>
<td>2-stage pipelined</td>
<td>50</td>
<td>1/25</td>
</tr>
</tbody>
</table>

Source: MIT 6.004 2019 L12
Representing pipelined circuits

```plaintext
Reg#(...) x <- mkReg(...);
Reg#(...) y <- mkReg(...);
rule p;
  let fv = F(x);
  let gv = G(y);
  let pv = H(fv, gv);
y <= pv;
endrule

Reg#(...) x <- mkReg(...);
Reg#(...) y <- mkReg(...);
Reg#(...) fv <- mkReg(...);
Reg#(...) gv <- mkReg(...);
rule p1;
  fv <= F(x);
  gv <= G(y);
endrule
rule p2;
  y <= H(fv, gv);
endrule
```

Does this work correctly?
Better implementation of pipelined circuits

\[
\begin{align*}
&\text{Reg}(\text{Bit}(8)) \ x \leftarrow \text{mkReg}(0); \\
&\text{Reg}(\text{Bit}(8)) \ y \leftarrow \text{mkReg}(0); \\
&\text{Reg}(\text{Bit}(8)) \ fv \leftarrow \text{mkReg}(0); \\
&\text{Reg}(\text{Bit}(8)) \ gv \leftarrow \text{mkReg}(0);
\end{align*}
\]

\begin{itemize}
  \item \textbf{rule} p1;
  \item \hspace{0.5cm} fv \leftarrow F(x);
  \item \hspace{0.5cm} gv \leftarrow G(x);
  \item \textbf{endrule}
  \\
  \item \textbf{rule} p2;
  \item \hspace{0.5cm} y \leftarrow H(fv, gv);
  \item \textbf{endrule}
\end{itemize}

\textbf{Issue:} In cycle 1, \( H \) calculates \( y \) using default values of \( fv \) and \( gv \).

\textbf{Issue:} If \( F \) or \( G \) blocks, \( p1 \) will not fire, and \( fv \) and \( gv \) will have stale values.

\textbf{But rule} \( p2 \) \textbf{will fire anyways, resulting in wrong data.}
Better implementation of pipelined circuits

Solution 1:
Lock-step pipeline

\[
\begin{align*}
\text{Reg}(\text{Bit#}(8)) & \ x \gets \text{mkReg}(0); \\
\text{Reg}(\text{Bit#}(8)) & \ y \gets \text{mkReg}(0); \\
\text{Reg}(\text{Bit#}(8)) & \ fv \gets \text{mkReg}(0); \\
\text{Reg}(\text{Bit#}(8)) & \ gv \gets \text{mkReg}(0); \\
\text{Reg}(\text{Bool}) & \ init \gets \text{mkReg}(\text{False}); \\
\text{rule} & \ p1; \\
\quad & \ fv \gets F(x); \\
\quad & \ gv \gets G(x); \\
\quad & \text{if} \ (\text{!init}) \ init \gets \text{True}; \\
\quad & \text{else} \ y \gets H(fv, gv); \\
\text{endrule}
\end{align*}
\]

Solution 2:
Elastic pipeline

\[
\begin{align*}
\text{Reg}(\text{Bit#}(8)) & \ x \gets \text{mkReg}(0); \\
\text{Reg}(\text{Bit#}(8)) & \ y \gets \text{mkReg}(0); \\
\text{FIFOF}(\text{Bit#}(8)) & \ fQ \gets \text{mkSizedFIFOF}(2); \\
\text{FIFOF}(\text{Bit#}(8)) & \ gQ \gets \text{mkSizedFIFOF}(2); \\
\text{rule} & \ p1; \\
\quad & \ fQ.\text{enq}(F(x)); \\
\quad & \ gQ.\text{enq}(G(x)); \\
\text{endrule} \\
\text{rule} & \ p2; \ // \text{fires only when data available} \\
\quad & \ fQ.\text{deq} ; \ gQ.\text{deq}; \\
\quad & \ y \gets H(fQ.\text{first}, \ gQ.\text{first}); \\
\text{endrule}
\end{align*}
\]
Aside: Even more elastic implementation

- x is pushed into a FIFO so that data is not dropped even if F or G blocks
  - y is also a FIFO for the same reason
- Is this a “better” implementation?
  - Maybe!
  - FIFOs use more chip space than registers
  - Depends on your workload...

```verilog
FIFOF#(Bit#(8)) xQ <- mkSizedFIFOF(2);
FIFOF#(Bit#(8)) yQ <- mkSizedFIFOF(2);
FIFOF#(Bit#(8)) fQ <- mkSizedFIFOF(2);
FIFOF#(Bit#(8)) gQ <- mkSizedFIFOF(2);

rule p1;
  xQ.deq;
  fQ.enq(F(xQ.first));
  gQ.enq(G(xQ.first));
endrule

rule p2;
  fQ.deq; gQ.deq;
  yQ.enq(H(fQ.first, gQ.first));
endrule
```
Pipeline conventions

Definition:
- A well-formed K-Stage Pipeline ("K-pipeline") is an acyclic circuit having exactly K registers on every path from an input to an output.
- A combinational circuit is thus a 0-stage pipeline.

Composition convention:
- Every pipeline stage, hence every K-Stage pipeline, has a register on its output (not on its input).

Clock period:
- The clock must have a period $t_{CLK}$ sufficient to cover the longest register to register propagation delay plus setup time.

K-pipeline latency = $K \times t_{CLK}$
K-pipeline throughput = $1 / t_{CLK}$

Source: MIT 6.004 2019 L12
Ill-formed pipelines

- Is the following circuit a K-stage pipeline? No

- Problem:
  - Some paths have different number of registers
  - Values from different input sets get mixed! -> Incorrect results
    - \( B(Y_{t-1}, A(X_t)) \) <- Mixing values from \( t \) and \( t-1 \)

Source: MIT 6.004 2019 L12
A pipelining methodology

- **Step 1:**
  - Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

- **Step 2:**
  - Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction.
  - These lines demarcate pipeline stages.

- **Step 3:**
  - Add a pipeline register at every point where a separating line crosses a connection.

Strategy: Try to break up high-latency elements, make each pipeline stage as low-latency as possible!

Source: MIT 6.004 2019 L12
Pipelining example

- 1-pipeline improves neither L nor T
- $T$ improved by breaking long combinational path, allowing faster clock
- Too many stages cost L, not improving $T$
- Back-to-back registers are sometimes needed for well-formed pipelines

<table>
<thead>
<tr>
<th></th>
<th>LATENCY</th>
<th>THROUGHPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>1-pipe:</td>
<td>4</td>
<td>1/4</td>
</tr>
<tr>
<td>2-pipe:</td>
<td>4</td>
<td>1/2</td>
</tr>
<tr>
<td>3-pipe:</td>
<td>6</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Source: MIT 6.004 2019 L12
Hierarchical pipelining

- Pipelined systems can be hierarchical
  - Replacing a slow combinational component with a k-pipe version may allow faster clock

- In the example:
  - 4-stage pipeline, T=1

Source: MIT 6.004 2019 L12
Sample pipelining problem

- Pipeline the following circuit for maximum throughput while minimizing latency.
  - Each module is labeled with its latency

What is the best latency and throughput achievable?
Sample pipelining problem

- $t_{CLK} = 4$
- $T = \frac{1}{4}$
- $L = 4 \times 4 = 16$
Benefits of an elastic pipeline

- Lock-step pipelines are great when modules are deterministic
  - Good for carefully scheduled circuits like a microprocessor
  - What if F or G blocks due to some internal logic?
  - What if H blocks due to some internal logic?
  - The whole pipeline is stuck!

- Elastic pipelines are good for decoupling modules
  - Internal implementation of modules have less effect on others

```verilog
rule p1;
fv <= F(x);
gv <= G(x);
if ( !init ) init <= True;
else y <= H(fv, gv);
endrule
```

Imagine if F and H are blocked on alternating cycles

```verilog
rule p1;
fQ.enq(F(x));
gQ.enq(G(x));
endrule
rule p2;
fQ.deq; gQ.deq;
y <= H(fQ.first, gQ.first);
endrule
```
Counting cycles: Benefits of an elastic pipeline

- Assume F and G are multi-cycle, internally pipelined modules
  - If we don’t know how many pipeline stages F or G has, how do we ensure correct results?

- Elastic pipeline allows correct results regardless of latency
  - If \( L(F) = L(G) \), enqueued data available at very next cycle (acts like single register)
  - If \( L(F) = L(G) + 1 \), FIFO acts like two pipelined registers
  - What if we made a 4-element FIFO, but \( L(F) = L(G) + 4 \)?
    - G will block! Results will still be correct!
    - ... Just slower! How slow?

L <- Latency in cycles
Measuring pipeline performance

- Latency of F is 3, latency of G is 1, and we have a 2-element FIFO
  - What would be the performance of this pipeline?

- One pipeline “bubble” every four cycles
  - Duty cycle of $\frac{3}{4}$!

*Animation*
Aside: Little’s law

- \( L = \lambda W \)
  - L: Number of requests in the system
  - \( \lambda \): Throughput
  - W: Latency
  - Imagine a DMV office! L: Number of booths. (Not number of chairs in the room)

In our pipeline example
- \( L = 3 \) (limited by pipeline depth of G)
- \( W = 4 \) (limited by pipeline depth of F)
- As a result: \( \lambda = \frac{3}{4} \)

How do we improve performance?
- Larger FIFO, or
- Replicate G! (round-robin use of G1 and G2)