Pipelining

Nachiket Kapre
nachiket@uwaterloo.ca
UNIVERSITY OF WATERLOO

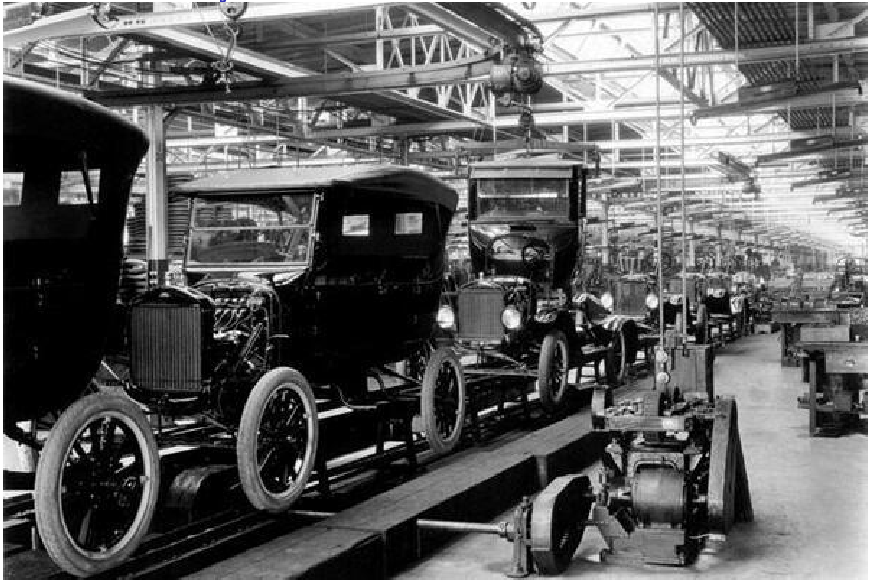
Outline

- Need for pipelining
  - CPU model (IPC, CPI)
- Latency and Throughput
- Pipelining Syntax
- Bubbles in a pipeline

Idea of pipelining

- Modern automated factories have *assembly lines*
- Each factory worker/robot specializes in one aspect of the manufacturing flow
- Allows us to design **high-throughput** pipelines of product manufacturing
- Compare: skilled craftsman who delivers entire product

Model T assembly



https://www.flickr.com/photos/bluespringsfordparts/14492600447/in/photostream/

Electronics assembly

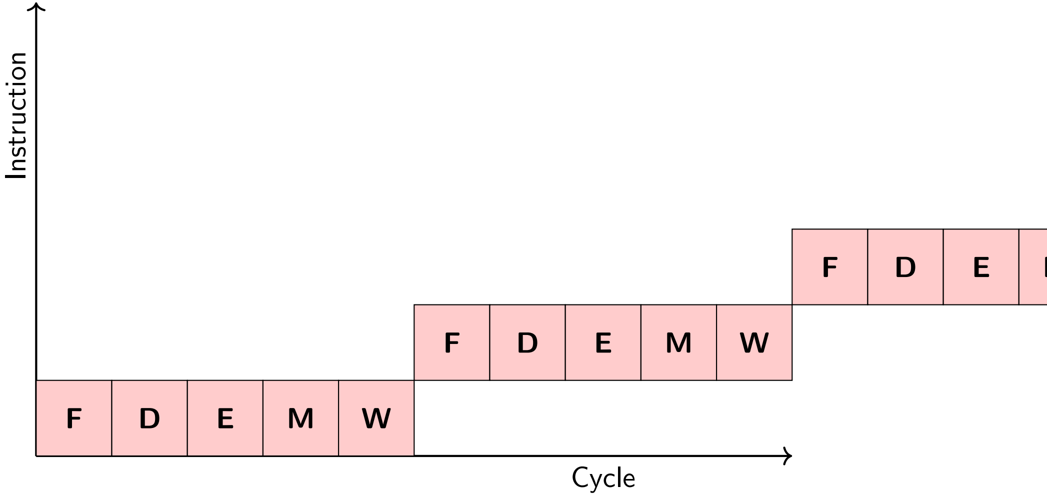
https://commons.wikimedia.org/wiki/File:Jinbei_production_line.jpg

Pipelining in Modern CPUs

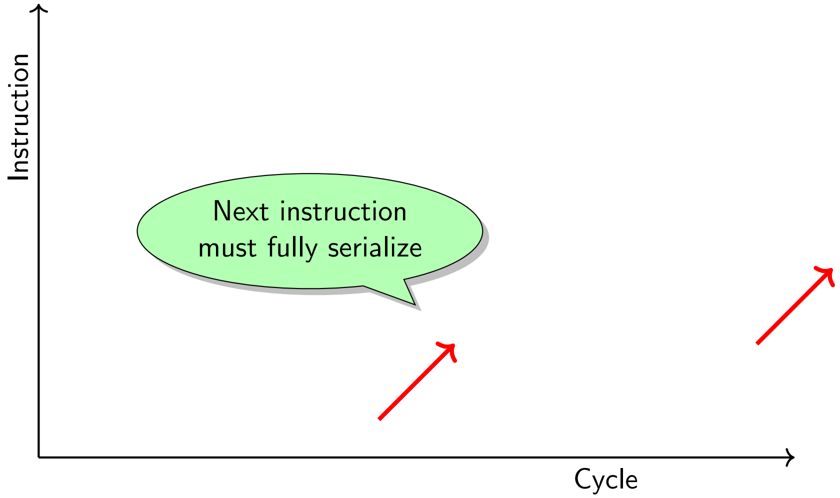
- C code compiled into a **sequence** of instructions (data movement + arithmetic)
- Simple CPU pipelines can pack entire instruction processing in a single cycle
  - **Fetch** → **Decode** → **Execute** → **Memory** → **Writeback**
- Better CPUs pipeline each stage into a cycle.
  - Pipelining allows **overlapping** of computation for different instructions in the same clock cycle
  - **But**, must check for dependencies/hazards.

Fetch
Decode
Execute
Memory
Write

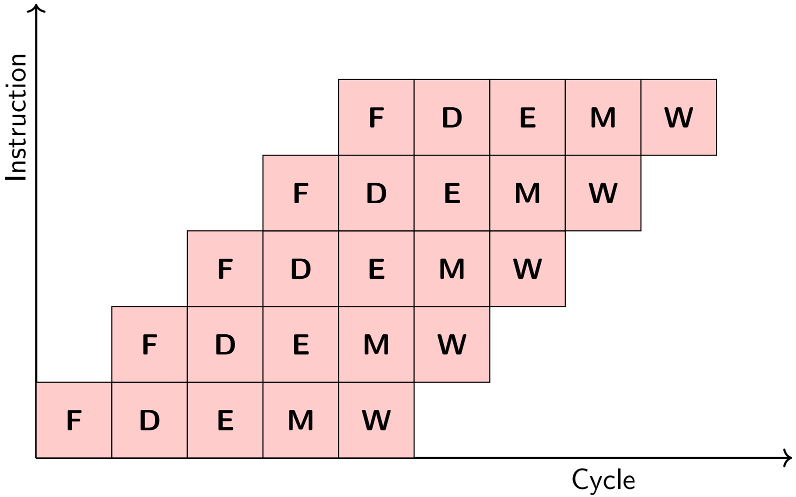
Execution Flow on a Unpipelined CPU



Execution Flow on a Unpipelined CPU



Execution Flow on a Pipelined CPU (no dependencies)



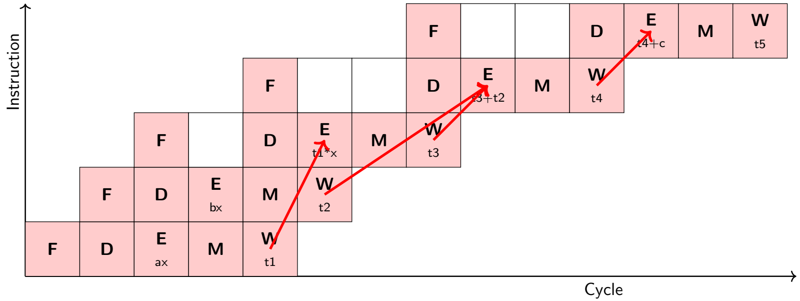
Advantages of Pipelining a CPU

- Faster clock frequency $\rightarrow$ fewer logic gate per stage
- Improved throughput of instructions retired
- **IPC** Instruction Per Cycle is $\uparrow$, **CPI** Clock cycles per instruction $\downarrow$
- Not a free lunch $\rightarrow$ pay extra in silicon area $+$ power
- Also, when dependencies occur, you have to **stall** the pipeline
  - Order of instruction execution is important
  - Previous instruction must finish before next **dependent** instruction is issued

Execution Flow on a Pipelined CPU (poly with dependencies)

```
poly:
        imull        %a, %x, %t1
        imull        %b, %x, %t2
        imull        %t1, %x, %t3
        addl        %t2, %t3, %t4
        addl        %c, %t4, %t5
        movl        %t5, %y
        ret
```

Execution Flow on a Pipelined CPU (with dependencies)



Understanding Pipelining in CPU

- CPU operation is internally pipelined (often, quite extensively)
- Dependencies cause **pipeline stalls** (or **bubbles**) in the pipeline
- **CPU has special hardware to track dependencies** $\rightarrow$
  - In-order processing: Each instruction checks if input operands used in previous k instruction window. k depends on how depth of CPU pipeline.
  - Out-of-order processing: Tomasulo's algorithm used for out-of-order processors where multiple instructions M can be issued together. Must track dependencies across k*M possible set of instructions.
- When designing your own custom hardware, you can reason about dependencies yourself!

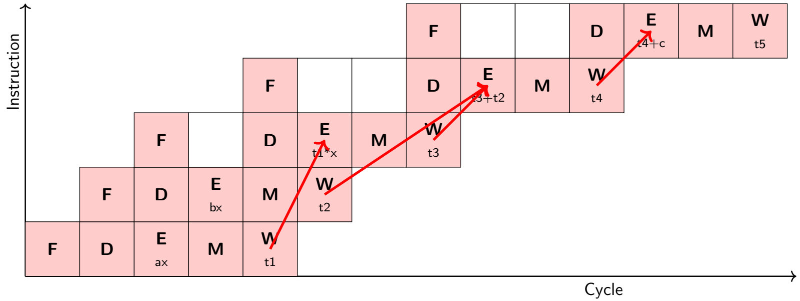
Pipelining Terminology

- When measuring CPU performance, we count
  - (1) *INST* total number of instructions retired/executed
  - (2) *CYC* total number of cycles required
- Metric of goodness
  - **IPC** Instructions Per Cycle $= \frac{INST}{CYC}$
  - Another metric is **CPI** Clock Cycles Per Instruction $= \frac{CYC}{INST}$
  - Thus, **IPC** $= 1/$**CPI**
- For unpipelined CPUs, **CPI** is depth of pipeline.
  - Here, the metric value does not change with dependencies
  - Paying the worst-case waiting time for each instruction
- For pipelined single-issue CPUs, **IPC** is 1 (best case).
  - Application dependencies will affect the metric in this case
  - If no dependency, IPC of 1 is possible. With dependencies, IPC will be < 1

Analyzing CPU performance (Unpipelined CPU)

- Observed counts
  - **CYC** = 5*k cycles
  - **INST** = k instructions
- Computation of performance metrics
  - **IPC** $= \frac{k}{5*k} = \frac{1}{5}$
  - **CPI** $= \frac{5*k}{k} = 5$
- Design is slow, but predictable

Execution Flow on a Pipelined CPU (with dependencies)



Analyzing CPU performance

- Observed counts
  - **CYC** $= 14$ cycles
  - **INST** $= 5$ instructions
- Computation of performance metrics
  - **IPC** $= \frac{5}{14}$
  - **CPI** $= \frac{14}{5}$
- Design is faster, but unpredictable (each application may run differently as per dependencies)

Effect of Pipelining

- **Deeper** pipelines $\rightarrow$ lot of register stages from input to output
- More registers $\rightarrow$ High Latency
- More registers $\rightarrow$ High Throughput
- More registers $\rightarrow$ Smaller clock period
- More registers $\rightarrow$ More area

Effect of Pipelining

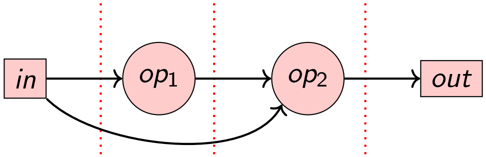
- Beyond a certain number of registers, clock period does not improve much $\rightarrow$ no more throughput improvements
- Only impact is higher latency and more area.
- **Challenge**: Find the sweet spot

Pipelining in poly

- Manually decompose the $a \times x^2 + b \times x + c$ into smaller operations
- 1–4 pipeline stages needed to hit granularity of individual operation
- Frequency is determined by the length of logic delays from input register to output register
- Note the delayed inputs – assume all inputs arrive at the same time. Important to delay signals to align their arrival at operator.

Pipelining Notation

- Show inputs and outputs with boxes, show arithmetic and logic operations in circles, show muxes with standard notation
- Pipelining stages are indicated with vertical dotted lines
  - If it helps you can label each stage by count (pipeline stage 1,2,3. . .)
- Remember, circuit edges that cross vertical pipeline edges **must** have a register for **each** cut.

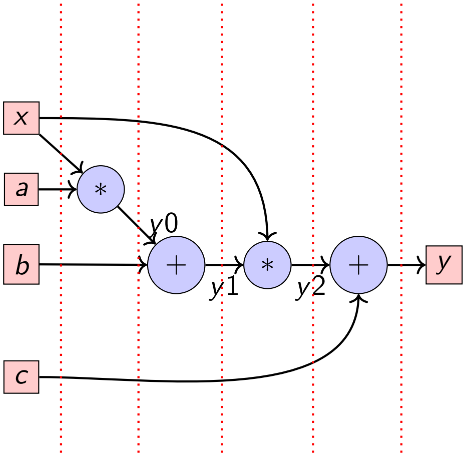


Pipelining Terminology

- **Throughput**: Rate at which circuit can consume inputs.
  - The highest throughput possible is 1, where you can push new inputs into the circuit every clock cycle.
  - If you can consume inputs once every two clocks, the throughput is $\frac{1}{2}$.
  - Throughput is not defined for combinational circuits. Need a periodic clock signal to define throughput.
- **Latency**: Time required for the first output to emerge from the circuit for a given input.
  - This is the end-to-end latency for one item of data to propagate from inputs of the circuit to its output.
  - Ideally, you want low latency, a value of 1 is minimum for a pipelined circuit.
  - A purely combinational output has a latency of 0
- **Clock Period**: Time separation between clock edges.
  - A faster frequency will results in small clock periods.
  - Ideally, you want to run your circuit as fast as required by the design specification.

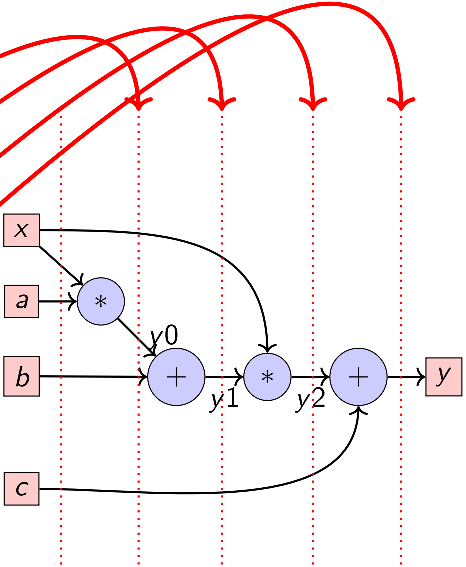
Pipelining poly circuit view

```
always @(posedge clk) begin
  if(rst) begin
    y0 <= {16{1'b0}};
    y1 <= {16{1'b0}};
    y2 <= {24{1'b0}};
    y <= {24{1'b0}};
    x_r1 <= {8{1'b0}};
    x_r2 <= {8{1'b0}};
    b_r1 <= {8{1'b0}};
    c_r1 <= {8{1'b0}};
    c_r2 <= {8{1'b0}};
    c_r3 <= {8{1'b0}};
  end else begin
    // stage 1
    y0 <= a_r * x_r;
    x_r1 <= x_r;
    b_r1 <= b_r;
    c_r1 <= c_r;
    // stage 2
    y1 <= y0 + b_r1;
    x_r2 <= x_r1;
    c_r2 <= c_r1;
    // stage 3
    y2 <= y1 * x_r2;
    c_r3 <= c_r2;
    // stage 4
    y <= y2 + c_r3;
  end
end
```



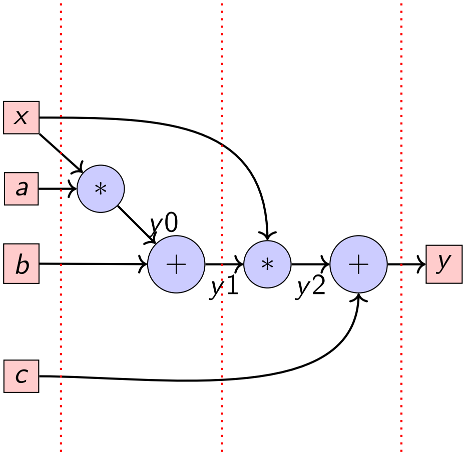
Pipelining poly circuit view

```
always @(posedge clk) begin
  if(rst) begin
    y0 <= {16{1'b0}};
    y1 <= {16{1'b0}};
    y2 <= {24{1'b0}};
    y <= {24{1'b0}};
    x_r1 <= {8{1'b0}};
    x_r2 <= {8{1'b0}};
    b_r1 <= {8{1'b0}};
    c_r1 <= {8{1'b0}};
    c_r2 <= {8{1'b0}};
    c_r3 <= {8{1'b0}};
  end else begin
    // stage 1
    y0 <= a_r * x_r;
    x_r1 <= x_r;
    b_r1 <= b_r;
    c_r1 <= c_r;
    // stage 2
    y1 <= y0 + b_r1;
    x_r2 <= x_r1;
    c_r2 <= c_r1;
    // stage 3
    y2 <= y1 * x_r2;
    c_r3 <= c_r2;
    // stage 4
    y <= y2 + c_r3;
  end
end
```



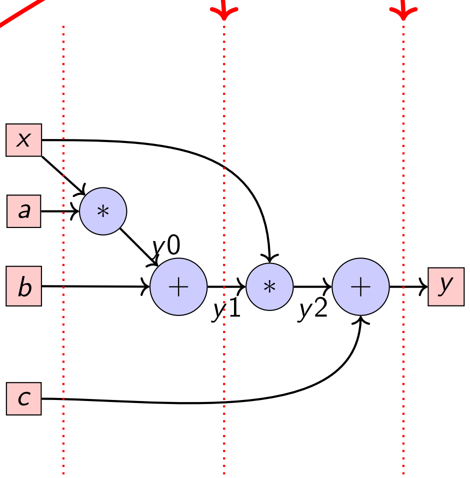
Pipelining poly circuit view

```
always @(posedge clk) begin
 if(rst) begin
  y1 <= {16{1'b0}};
  y <= {24{1'b0}};
  x_r1 <= {8{1'b0}};
  c_r1 <= {8{1'b0}};
 end else begin
  // stage 1
  y1 <= a_r * x_r + b_r;
  x_r1 <= x_r;
  c_r1 <= c_r;
  // stage 2
  y <= y1 * x_r1 + c_r1;
 end
end
```



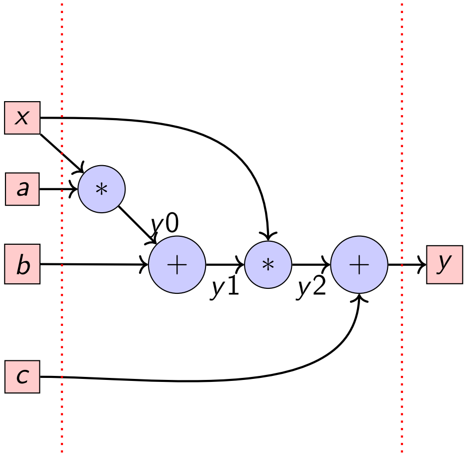
Pipelining poly circuit view

```
always @(posedge clk) begin
 if(rst) begin
  y1 <= {16{1'b0}};
  y <= {24{1'b0}};
  x_r1 <= {8{1'b0}};
  c_r1 <= {8{1'b0}};
 end else begin
  // stage 1
  y1 <= a_r * x_r + b_r;
  x_r1 <= x_r;
  c_r1 <= c_r;
  // stage 2
  y <= y1 * x_r1 + c_r1;
 end
end
```



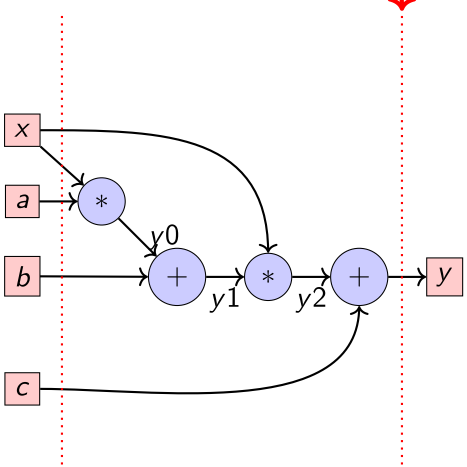
Pipelining poly circuit view

```
always @(posedge clk) begin
 if(rst) begin
  y <= {24{1'b0}};
 end else begin
  // stage 1
  y <= (a_r * x_r + b_r) * x_r + c_r;
 end
```



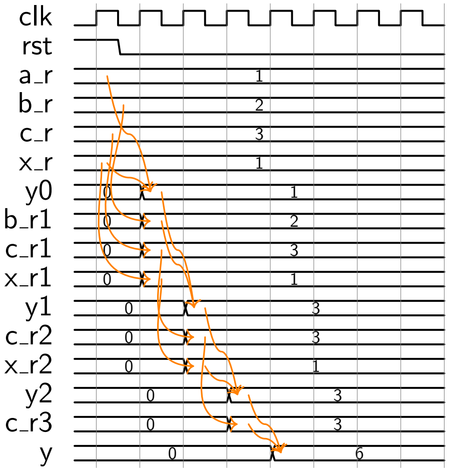
Pipelining poly circuit view

```
always @(posedge clk) begin
 if(rst) begin
  y <= {24{1'b0}};
 end else begin
  // stage 1
  y <= (a_r * x_r + b_r) * x_r + c_r;
 end
```



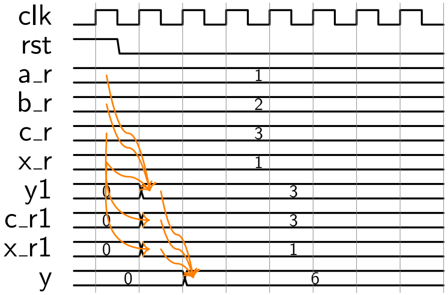
Pipelining Waveforms (4-Stage Pipeline)

```
always @(posedge clk) begin
  if(rst) begin
    y0 <= {16{1'b0}};
    y1 <= {16{1'b0}};
    y2 <= {24{1'b0}};
    y <= {24{1'b0}};
    x_r1 <= {8{1'b0}};
    x_r2 <= {8{1'b0}};
    b_r1 <= {8{1'b0}};
    c_r1 <= {8{1'b0}};
    c_r2 <= {8{1'b0}};
    c_r3 <= {8{1'b0}};
  end else begin
    // stage 1
    y0 <= a_r * x_r;
    x_r1 <= x_r;
    b_r1 <= b_r;
    c_r1 <= c_r;
    // stage 2
    y1 <= y0 + b_r1;
    x_r2 <= x_r1;
    c_r2 <= c_r1;
    // stage 3
    y2 <= y1 * x_r2;
    c_r3 <= c_r2;
    // stage 4
    y <= y2 + c_r3;
  end
end
```



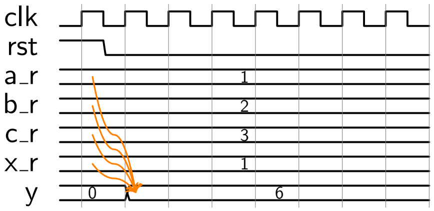
Pipelining Waveforms (2-Stage Pipeline)

```
always @(posedge clk) begin
 if(rst) begin
  y1 <= {16{1'b0}};
  y <= {24{1'b0}};
  x_r1 <= {8{1'b0}};
  c_r1 <= {8{1'b0}};
 end else begin
  // stage 1
  y1 <= a_r * x_r + b_r;
  x_r1 <= x_r;
  c_r1 <= c_r;
  // stage 2
  y <= y1 * x_r1 + c_r1;
 end
end
```



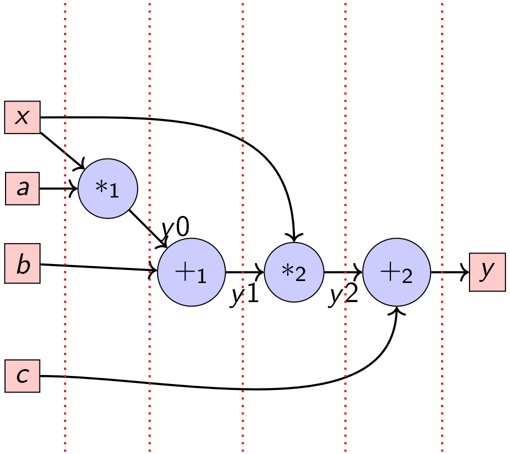
Pipelining Waveforms (1-Stage Pipeline)

```
always @(posedge clk) begin
 if(rst) begin
  y <= {24{1'b0}};
 end else begin
  // stage 1
  y <= (a_r * x_r + b_r) * x_r + c_r;
 end
```



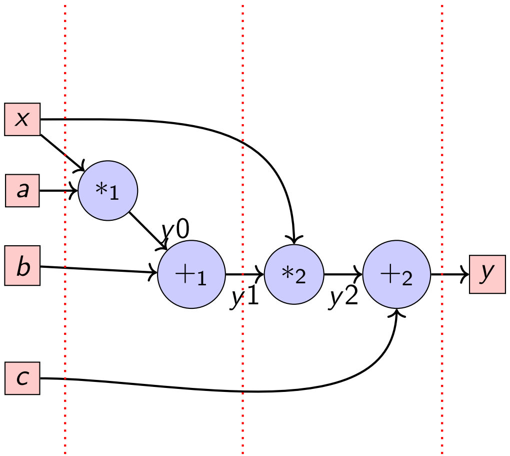
Pipelining poly analysis

- Latency = 5
- Throughput = 1
- Cost = (8+8+8+8) + (16+8+8+8) + (16+8+8) + (24+8) + 24 = 160
- Each circuit edge that crosses vertical line requires a FF/register
- Clock Period = $\max(T_{*_1}, T_{+_1}, T_{*_2}, T_{+_2})$
- General expression is max over all FF→FF paths



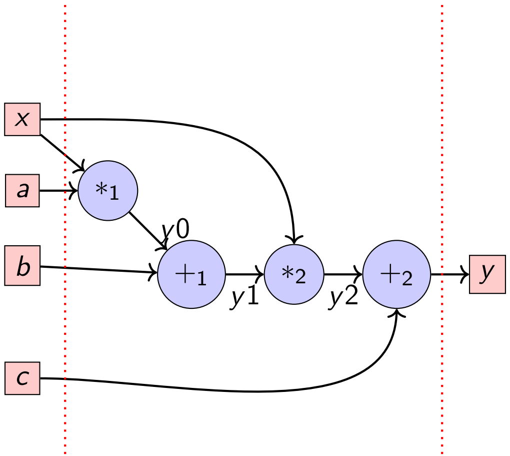
Pipelining poly analysis

- Latency $= 3$
- Throughput $= 1$
- Cost $= (8{+}8{+}8{+}8) + (16{+}8{+}8) + 24 = 88$
- Clock Period $= \max(T_{*_1} + T_{+_1}, T_{*_2} + T_{+_2})$



Pipelining poly analysis

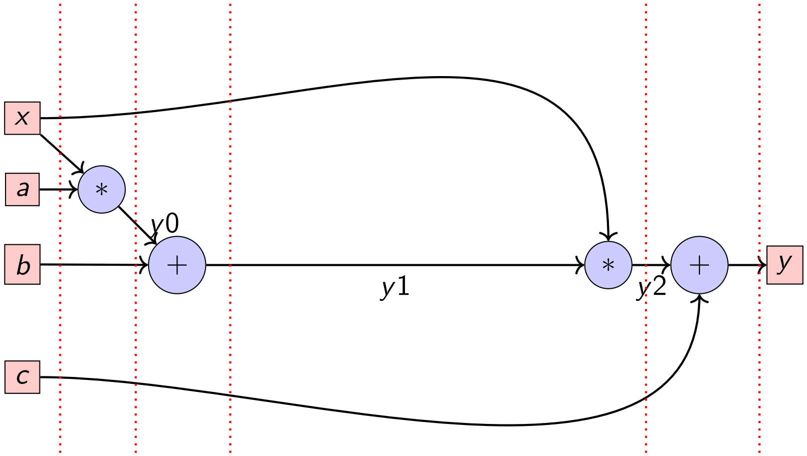
- Latency = 2
- Throughput = 1
- Cost = (8+8+8+8) + 24 = 56
- Clock Period = $T_{*_1} + T_{+_1} + T_{*_2} + T_{+_2}$
- **Note**: Throughput stayed 1 in all cases



Wire Pipelining

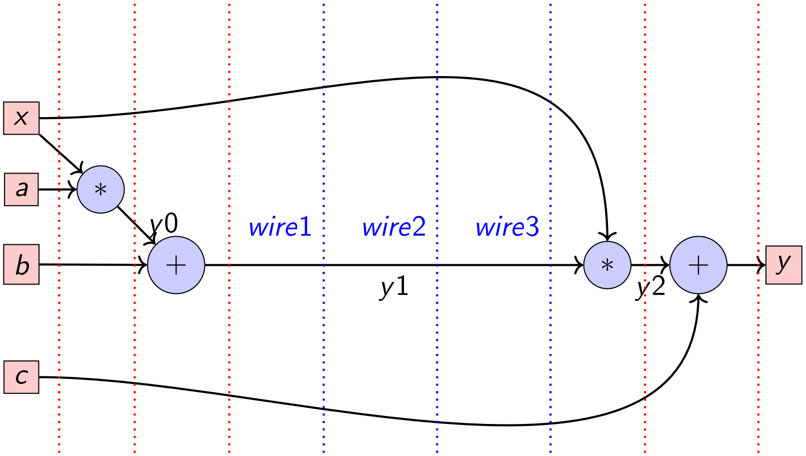
- On modern FPGAs and ASICs, most of the delay is in the wire, and not logic
- For instance, gates/LUTs may take 0.2–0.3 ns while a wire connecting the output to next gate make take 1–2 ns.
- Increasingly, we must extend the idea of pipelining to wires.

Wire pipelining poly circuit view



Clock Period =

Wire pipelining poly circuit view



Clock Period =

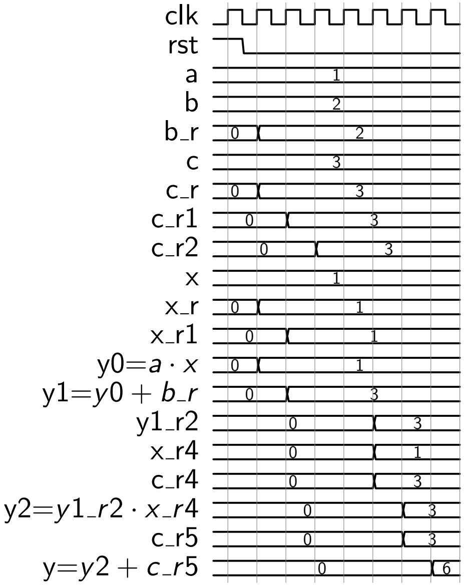
Wire Pipelining Code

```
// state 1
y0 <= a * x;
x_r <= x;
b_r <= b;
c_r <= c;
// stage 2
y1 <= y0 + b_r;
x_r1 <= x_r;
c_r1 <= c_r;
// wire delays
y1_r <= y1;
x_r2 <= x_r1;
c_r2 <= c_r1;
y1_r1 <= y1_r;
x_r3 <= x_r2;
c_r3 <= c_r2;
y1_r2 <= y1_r1;
x_r4 <= x_r3;
c_r4 <= c_r3;
// stage 3
y2 <= y1_r2 * x_r4;
c_r5 <= c_r4;
// stage 4
y <= y2 + c_r5;
```

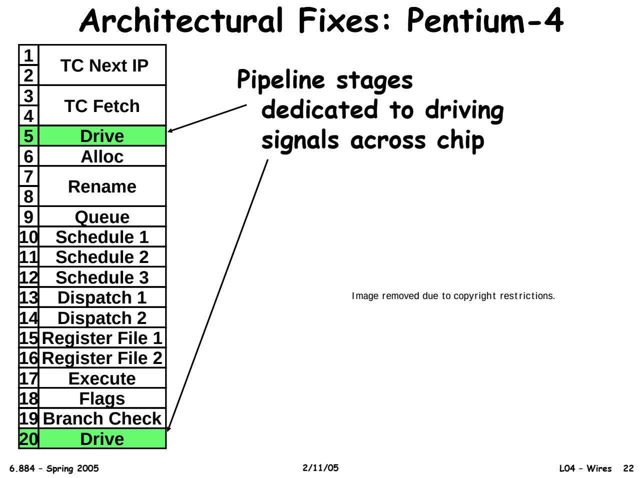
- In Verilog, you need to add dummy pipeline stages $\rightarrow$ they do no compute, but provide freedom to the FPGA CAD tools to split wires
- Wirelength is only known after full place-and-route.
- Synthesis tools aim to *estimate* wire lengths and delays, but a few iterative compilations are required regardless.
  - Similar to profile-guided optimization in gcc
- **Impact**:
  - Must design **parametric** hardware
  - Retiming can automate this

Wire Pipelining Waveform

```
// state 1
y0 <= a * x;
x_r <= x;
b_r <= b;
c_r <= c;
// stage 2
y1 <= y0 + b_r;
x_r1 <= x_r;
c_r1 <= c_r;
// wire delays
y1_r <= y1;
x_r2 <= x_r1;
c_r2 <= c_r1;
y1_r1 <= y1_r;
x_r3 <= x_r2;
c_r3 <= c_r2;
y1_r2 <= y1_r1;
x_r4 <= x_r3;
c_r4 <= c_r3;
// stage 3
y2 <= y1_r2 * x_r4;
c_r5 <= c_r4;
// stage 4
y <= y2 + c_r5;
```



DRIVE stage in the Pentium-4 pipeline



Architectural Fixes: Pentium-4

1	TC Next IP
2	
3	TC Fetch
4	
5	Drive
6	Alloc
7	Rename
8	
9	Queue
10	Schedule 1
11	Schedule 2
12	Schedule 3
13	Dispatch 1
14	Dispatch 2
15	Register File 1
16	Register File 2
17	Execute
18	Flags
19	Branch Check
20	Drive

Pipeline stages dedicated to driving signals across chip

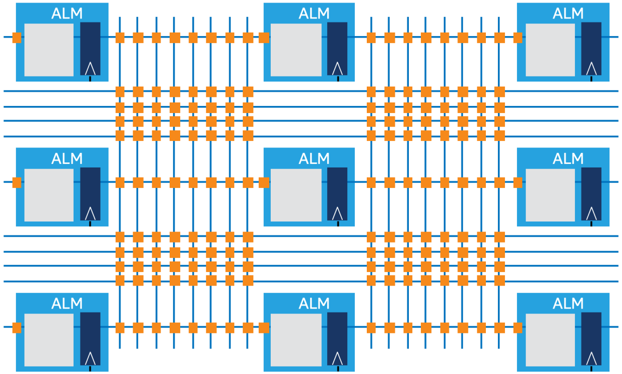
Image removed due to copyright restrictions.

6.884 - Spring 2005 2/11/05 L04 - Wires 22

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-884-complex-digital-systems-spring-2005/lecture-notes/

l04_wires.pdf

Extensive wire pipelinining in Intel Hyperflex FPGA



- Registers are available in every routing segment
- Registers are available on all block inputs (ALM, M20K blocks, DSP blocks, and I/O cells)

`https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/`

`wp-01231-understanding-how-hyperflex-architecture-enables-high-performance-systems.pdf`

Pipeline bubbles

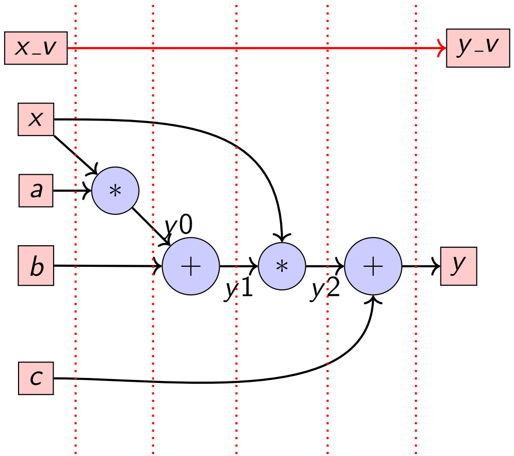
- In water pipes, bubbles arise due to air intake $\rightarrow$ bubble smoothly rides withe flow and is released.
- In CPUs, processing pipeline **bubbles** arise due to stalls (which in turn are due to data dependencies).
- In custom datapaths, bubbles arise due to resource sharing (next lecture), or absence of input data.
  - Resource sharing is known upfront when designing the hardware
  - Data arrival uncertainty is now known at compile/design time
- **Key**: A bubble is a NULL operation through the hardware pipeline. Outputs of the pipeline stage when the bubble is passing through is ignored. Bubble moves through the pipeline in LATENCY cycles.

Implementing bubbles

- Bubbles are simply **invalid** data in a pipeline
- Can be trivially handled with a special **valid** indication
- For resource sharing , **valid** signal generated by state machine
- For absent data, **valid** signal must be dynamically determined
  - Tag input with valid, and output with valid
  - Generate a shift register with identical delay as pipeline latency
  - Feed input valid into the shift register, and connect output valid to output of shift register
- Must handle multiplexers (if present) carefully $\rightarrow$ need a separate shift register for each branch and select input, and combine the valids based on select appropriately

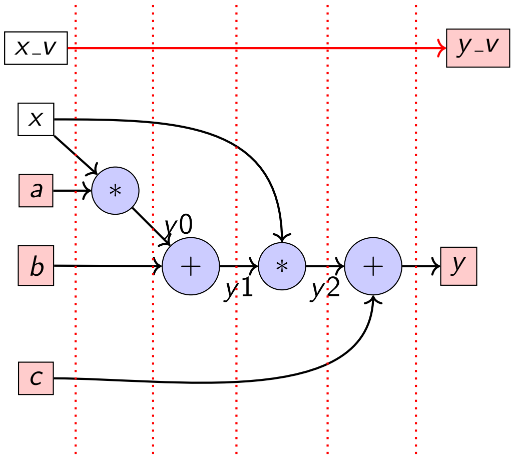
Implementing bubbles

```
always @(posedge clk) begin: valid
 if(rst) begin
  v <= {3{1'b0}};
  y_v <= 1'b0;
 end else begin
  v[0] <= x_v; //
  v[1] <= v[0];
  v[2] <= v[1];
  v[3] <= v[2];
  y_v <= v[3]; //
 end
end
```



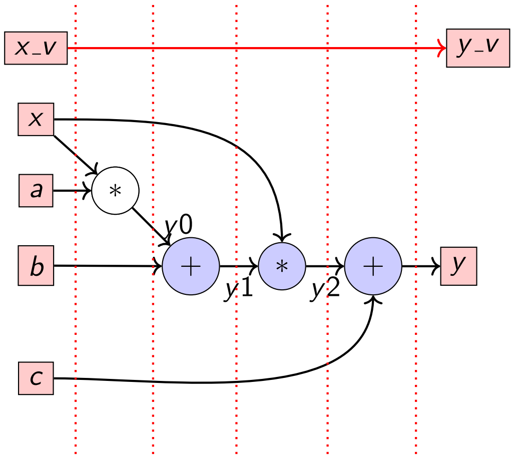
Implementing bubbles

```
always @(posedge clk) begin: valid
 if(rst) begin
  v <= {3{1'b0}};
  y_v <= 1'b0;
 end else begin
  v[0] <= x_v; //
  v[1] <= v[0];
  v[2] <= v[1];
  v[3] <= v[2];
  y_v <= v[3]; //
 end
end
```



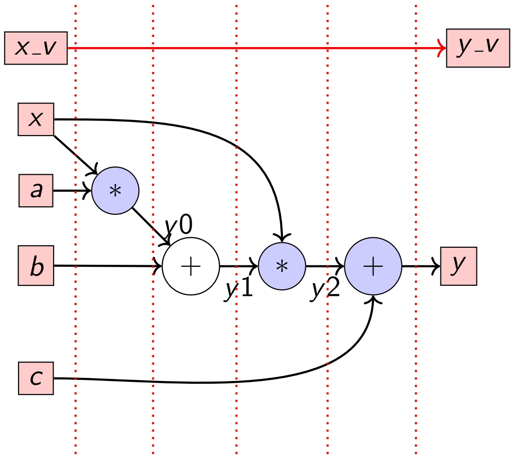
Implementing bubbles

```
always @(posedge clk) begin: valid
 if(rst) begin
  v <= {3{1'b0}};
  y_v <= 1'b0;
 end else begin
  v[0] <= x_v; //
  v[1] <= v[0];
  v[2] <= v[1];
  v[3] <= v[2];
  y_v <= v[3]; //
 end
end
```



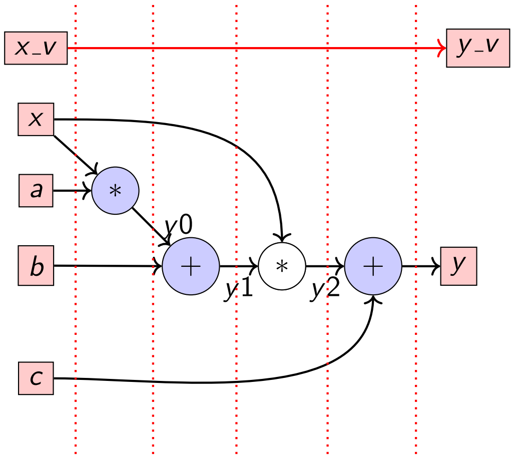
Implementing bubbles

```
always @(posedge clk) begin: valid
 if(rst) begin
  v <= {3{1'b0}};
  y_v <= 1'b0;
 end else begin
  v[0] <= x_v; //
  v[1] <= v[0];
  v[2] <= v[1];
  v[3] <= v[2];
  y_v <= v[3]; //
 end
end
```



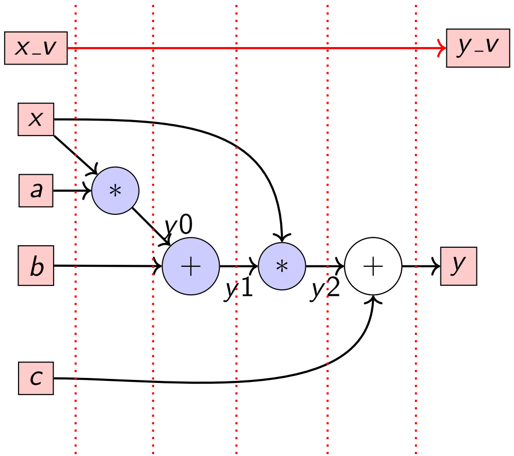
Implementing bubbles

```
always @(posedge clk) begin: valid
 if(rst) begin
  v <= {3{1'b0}};
  y_v <= 1'b0;
 end else begin
  v[0] <= x_v; //
  v[1] <= v[0];
  v[2] <= v[1];
  v[3] <= v[2];
  y_v <= v[3]; //
 end
end
```



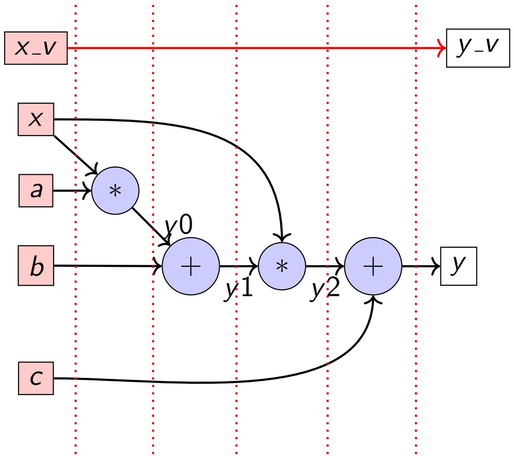
Implementing bubbles

```
always @(posedge clk) begin: valid
 if(rst) begin
  v <= {3{1'b0}};
  y_v <= 1'b0;
 end else begin
  v[0] <= x_v; //
  v[1] <= v[0];
  v[2] <= v[1];
  v[3] <= v[2];
  y_v <= v[3]; //
 end
end
```



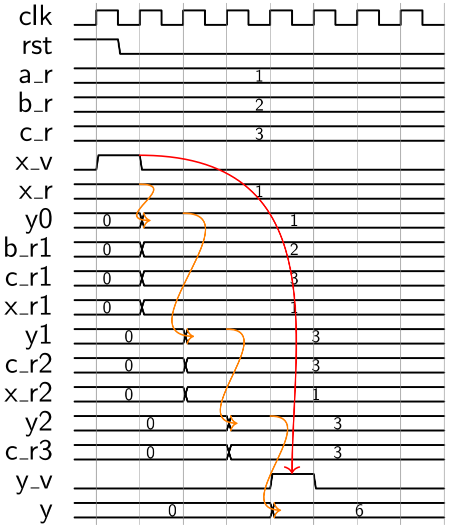
Implementing bubbles

```
always @(posedge clk) begin: valid
 if(rst) begin
  v <= {3{1'b0}};
  y_v <= 1'b0;
 end else begin
  v[0] <= x_v; //
  v[1] <= v[0];
  v[2] <= v[1];
  v[3] <= v[2];
  y_v <= v[3]; //
 end
end
```



Waveform for bubbles

```
always @(posedge clk) begin: valid
 if(rst) begin
  v <= {3{1'b0}};
  y_v <= 1'b0;
 end else begin
  v[0] <= x_v; //
  v[1] <= v[0];
  v[2] <= v[1];
  v[3] <= v[2];
  y_v <= v[3]; //
 end
end
```



Wrapup

- Datapath pipelining useful to improve throughput at the expense of latency
- Unlike CPU pipelines, custom datapath pipelines allow a designer to choose throughput, latency, clock period combinations to meet system requirements
- Wire delays more important in modern chips $\rightarrow$ pipelining must consider wires.
- Bubbles in the input must be handled carefully to avoid corrupting output data
- **Question:** Can we pipeline any circuit?
  - Fundamental limit to how much pipelining is possible $\rightarrow$ single LUT
  - Feedforward circuits can always be pipelined