Static Scheduling

Nachiket Kapre
nachiket@uwaterloo.ca

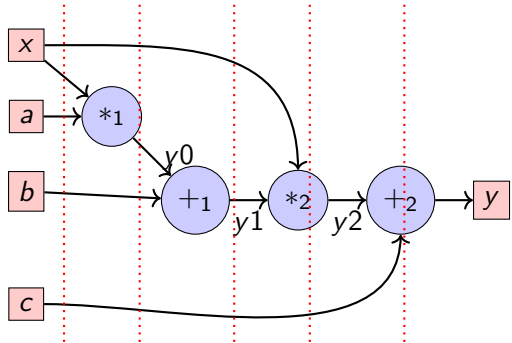
Outline

- Need for resource sharing
  - Design constraints (few resources, few IOs)
  - CPU model (shared ALU)
- Dataflow Analysis
  - Implementation Task
  - Optimization Task
- RTL Implementation

Need for resource sharing

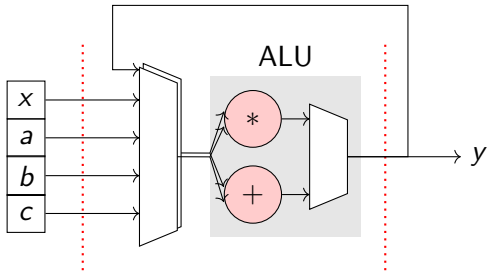
- **Efficiency**: Hardware design on small budgets often have to use limited chip resources
- **Performance**: Even when hardware costs are not a constraint, packing multiple operations into small chip area $\rightarrow$ saves on wire delay costs
- **Portability**: If the list of resource shared operators are known, computation becomes portable. CPU ALUs are the simplest example of resource sharing.

Fully Spatial (Parallel) Design



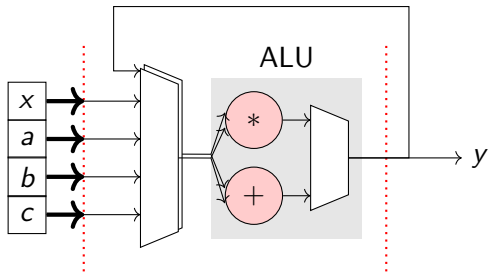
- You can use as many adders and multiplers as you want
- Free to use as many pipeline stages as required
- For now, we skip registering inputs
- Cost is no concern $\rightarrow$ Throughput=1, Latency=5, $\downarrow$ Clock Period = $\max(T_{*_1}, T_{+_1}, T_{*_2}, T_{+_2})$.

Fully Sequential Design



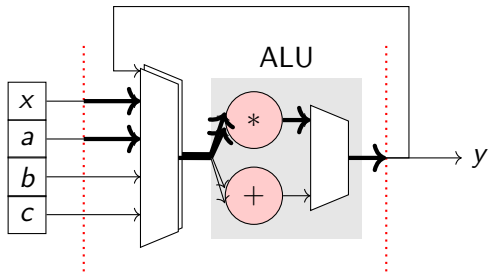
- One of each – arithmetic operator, register, IO port
- Sharing enabled with multiplexers
- Low-cost design $\rightarrow$ Latency $= 5$ (still), but Throughput$=\frac{1}{4}$, as we have to wait for y to be pushed out first
- Clock Period can be made small $\rightarrow \max(T_{imux} + T_{*} + T_{omux}, T_{imux} + T_{+} + T_{omux})$

Fully Sequential Design



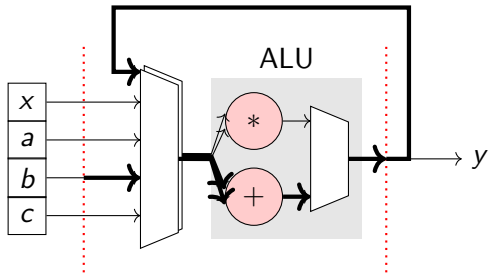
- 1st Cycle $\rightarrow a|b|c|x\_r <= a, b, c, x$
- 2nd Cycle $\rightarrow t1 <= a\_r \times x\_r$
- 3rd Cycle $\rightarrow t2 <= t1 + b\_r$
- 4th Cycle $\rightarrow t3 <= t2 \times x\_r$
- 5th Cycle $\rightarrow y <= t3 + c\_r;\ a|b|c|x\_r <= a, b, c, x;$
- 6th Cycle $\rightarrow t1 <= a\_r \times x\_r$!!

Fully Sequential Design



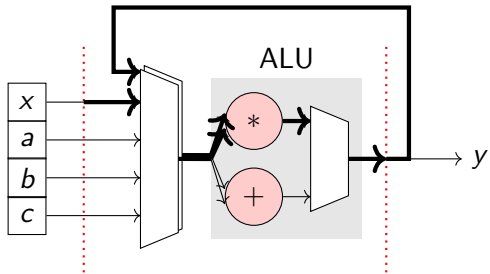
- 1st Cycle $\rightarrow a|b|c|x\_r <= a, b, c, x$
- 2nd Cycle $\rightarrow t1 <= a\_r \times x\_r$
- 3rd Cycle $\rightarrow t2 <= t1 + b\_r$
- 4th Cycle $\rightarrow t3 <= t2 \times x\_r$
- 5th Cycle $\rightarrow y <= t3 + c\_r;\ a|b|c|x\_r <= a, b, c, x;$
- 6th Cycle $\rightarrow t1 <= a\_r \times x\_r$!!

Fully Sequential Design



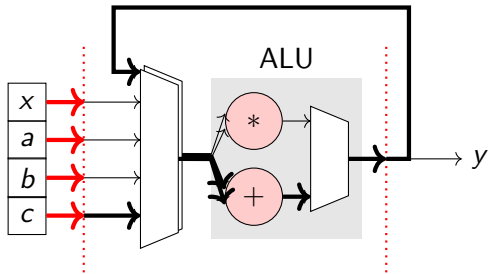
- 1st Cycle $\rightarrow a|b|c|x\_r <= a, b, c, x$
- 2nd Cycle $\rightarrow t1 <= a\_r \times x\_r$
- 3rd Cycle $\rightarrow t2 <= t1 + b\_r$
- 4th Cycle $\rightarrow t3 <= t2 \times x\_r$
- 5th Cycle $\rightarrow y <= t3 + c\_r;\ a|b|c|x\_r <= a, b, c, x;$
- 6th Cycle $\rightarrow t1 <= a\_r \times x\_r$!!

Fully Sequential Design



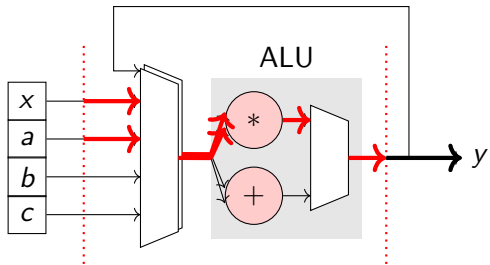
- 1st Cycle $\rightarrow a|b|c|x\_r <= a, b, c, x$
- 2nd Cycle $\rightarrow t1 <= a\_r \times x\_r$
- 3rd Cycle $\rightarrow t2 <= t1 + b\_r$
- 4th Cycle $\rightarrow t3 <= t2 \times x\_r$
- 5th Cycle $\rightarrow y <= t3 + c\_r;\ a|b|c|x\_r <= a, b, c, x;$
- 6th Cycle $\rightarrow t1 <= a\_r \times x\_r$!!

Fully Sequential Design



- 1st Cycle $\rightarrow a|b|c|x\_r <= a, b, c, x$
- 2nd Cycle $\rightarrow t1 <= a\_r \times x\_r$
- 3rd Cycle $\rightarrow t2 <= t1 + b\_r$
- 4th Cycle $\rightarrow t3 <= t2 \times x\_r$
- 5th Cycle $\rightarrow y <= t3 + c\_r;\ a|b|c|x\_r <= a, b, c, x;$
- 6th Cycle $\rightarrow t1 <= a\_r \times x\_r$!!

Fully Sequential Design



- 1st Cycle $\rightarrow a|b|c|x\_r <= a, b, c, x$
- 2nd Cycle $\rightarrow t1 <= a\_r \times x\_r$
- 3rd Cycle $\rightarrow t2 <= t1 + b\_r$
- 4th Cycle $\rightarrow t3 <= t2 \times x\_r$
- 5th Cycle $\rightarrow y <= t3 + c\_r;\ a|b|c|x\_r <= a, b, c, x;$
- 6th Cycle $\rightarrow t1 <= a\_r \times x\_r$!!

```
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
```

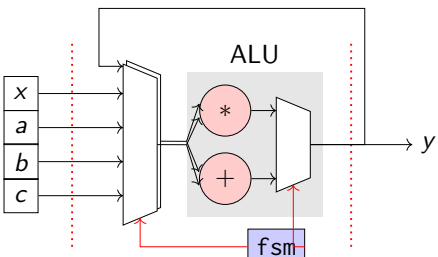
```
always @(posedge clk) begin
 if(rst) begin
  count <= {3{1'b0}};
  y_r <= {24{1'b0}};
 end else begin
  if(count == 0) begin
   count <= count + 1;
   y_r[15:0] <= a_r * x_r;
  end
  else if(count == 1) begin
   count <= count + 1;
   y_r <= y_r + b_r;
  end
  else if(count == 2) begin
   count <= count + 1;
   y_r <= y_r[15:0] * x_r;
  end
  else if(count == 3) begin
   count <= {3{1'b0}};
   y_r <= y_r + c_r;
  end
 end
end

assign y = y_r;
```

```
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
```

```
always @(posedge clk) begin
 if(rst) begin
  y_r <= {24{1'b0}};
 end else begin
  case(count)
   2'b00 : begin
    y_r[15:0] <= a * x;
   end
   2'b01 : begin
    y_r <= y_r + b;
   end
   2'b10 : begin
    y_r <= y_r[15:0] * x;
   end
   2'b11 : begin
    y_r <= y_r + c;
   end
   default : begin
    y_r <= {24{1'b0}};
   end
  endcase
 end
end

assign y = y_r;
```

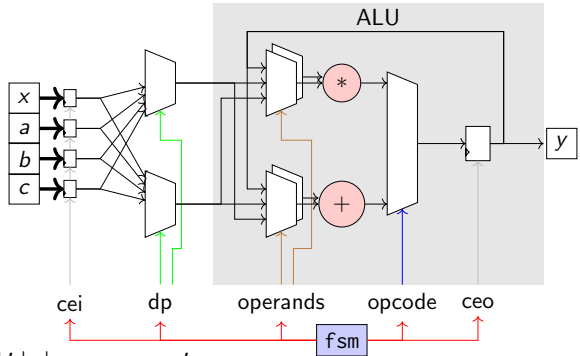
```
always @(posedge clk) begin
 if(rst) begin
  y0 <= {16{1'b0}};
  y1 <= {16{1'b0}};
  y2 <= {24{1'b0}};
  y <= {24{1'b0}};
  x_r1 <= {8{1'b0}};
  x_r2 <= {8{1'b0}};
  b_r1 <= {8{1'b0}};
  c_r1 <= {8{1'b0}};
  c_r2 <= {8{1'b0}};
  c_r3 <= {8{1'b0}};
 end else begin
  // stage 1
  y0 <= a_r * x_r;
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

```
// imux
assign mux0 = (count == 1) ? {4'h0,a_r} : y_r;
assign mux1 = (count == 1) ? {4'h0,x_r} :
       (count == 2) ? {4'h0,b_r} :
       (count == 3) ? {4'h0,x_r} :
                      {4'h0,c_r};
// arithmetic datapath
assign add0 = mux0 + mux1;
assign mult0 = mux0[15:0] * mux1[7:0];

always @(posedge clk) begin
 if(rst) begin
  y_r <= {24{1'b0}};
 end else begin
  case(count)
   2'b01,2'b11 : begin
    y_r <= mult0;
   end
   2'b10,2'b00 : begin
    y_r <= add0;
   end
   default : begin
    y_r <= {24{1'b0}};
   end
  endcase
 end
end

assign y = y_r;
```



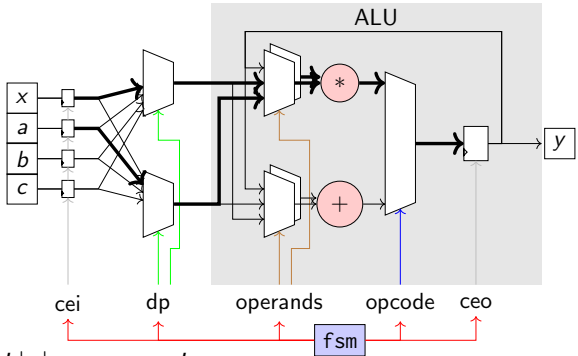

```
// imux
assign mux0 = (count == 1) ? {4'h0,a_r} : y_r;
assign mux1 = (count == 1) ? {4'h0,x_r} :
       (count == 2) ? {4'h0,b_r} :
       (count == 3) ? {4'h0,x_r} :
                      {4'h0,c_r};
// arithmetic datapath
assign add0 = mux0 + mux1;
assign mult0 = mux0[15:0] * mux1[7:0];

always @(posedge clk) begin
 if(rst) begin
  y_r <= {24{1'b0}};
 end else begin
  case(count)
   2'b01,2'b11 : begin
    y_r <= mult0;
   end
   2'b10,2'b00 : begin
    y_r <= add0;
   end
   default : begin
    y_r <= {24{1'b0}};
   end
  endcase
 end
end

assign y = y_r;
```

Sequential with IO limits



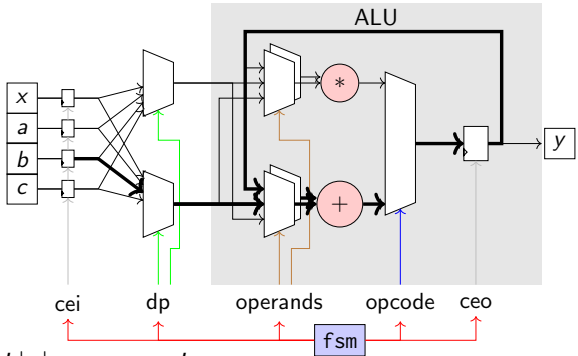
- 1st Cycle $\rightarrow a|b|c|x\_r <= a, b, c, x$
- 2nd Cycle $\rightarrow t1 <= a\_r \times x\_r$
- 3rd Cycle $\rightarrow t2 <= t1 + b\_r$
- 4th Cycle $\rightarrow t3 <= t2 \times x\_r$
- 5th Cycle $\rightarrow y <= t3 + c\_r,\ a|b|c|x\_r <= a, b, c, x$
- 6th Cycle $\rightarrow t1 <= a\_r \times x\_r$!!

Sequential with IO limits



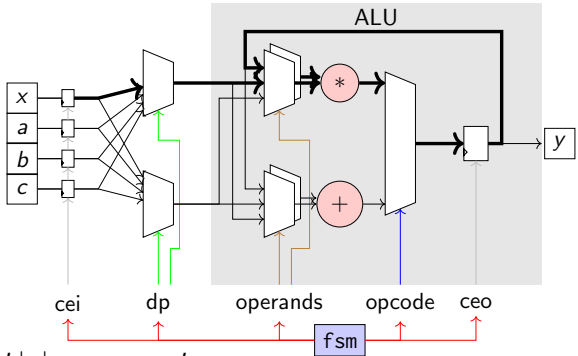
- 1st Cycle $\rightarrow a|b|c|x\_r <= a, b, c, x$
- 2nd Cycle $\rightarrow t1 <= a\_r \times x\_r$
- 3rd Cycle $\rightarrow t2 <= t1 + b\_r$
- 4th Cycle $\rightarrow t3 <= t2 \times x\_r$
- 5th Cycle $\rightarrow y <= t3 + c\_r,\ a|b|c|x\_r <= a, b, c, x$
- 6th Cycle $\rightarrow t1 <= a\_r \times x\_r$!!

Sequential with IO limits



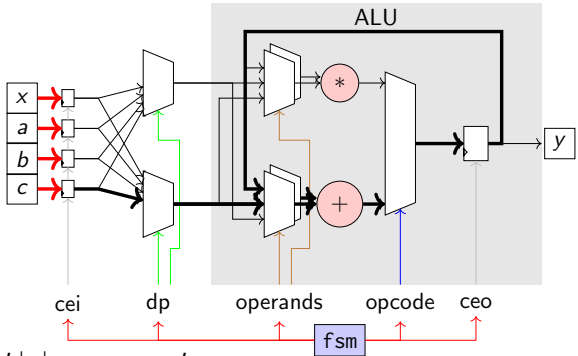
- 1st Cycle $\rightarrow a|b|c|x\_r <= a, b, c, x$
- 2nd Cycle $\rightarrow t1 <= a\_r \times x\_r$
- 3rd Cycle $\rightarrow t2 <= t1 + b\_r$
- 4th Cycle $\rightarrow t3 <= t2 \times x\_r$
- 5th Cycle $\rightarrow y <= t3 + c\_r,\ a|b|c|x\_r <= a, b, c, x$
- 6th Cycle $\rightarrow t1 <= a\_r \times x\_r$!!

Sequential with IO limits



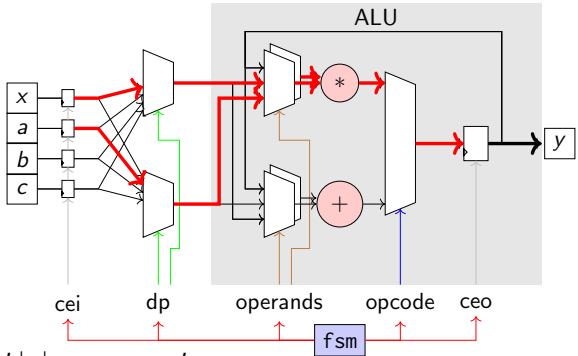
- 1st Cycle $\rightarrow a|b|c|x\_r <= a, b, c, x$
- 2nd Cycle $\rightarrow t1 <= a\_r \times x\_r$
- 3rd Cycle $\rightarrow t2 <= t1 + b\_r$
- 4th Cycle $\rightarrow t3 <= t2 \times x\_r$
- 5th Cycle $\rightarrow y <= t3 + c\_r,\ a|b|c|x\_r <= a, b, c, x$
- 6th Cycle $\rightarrow t1 <= a\_r \times x\_r$!!

Sequential with IO limits



- 1st Cycle $\rightarrow a|b|c|x\_r <= a, b, c, x$
- 2nd Cycle $\rightarrow t1 <= a\_r \times x\_r$
- 3rd Cycle $\rightarrow t2 <= t1 + b\_r$
- 4th Cycle $\rightarrow t3 <= t2 \times x\_r$
- 5th Cycle $\rightarrow y <= t3 + c\_r,\ a|b|c|x\_r <= a, b, c, x$
- 6th Cycle $\rightarrow t1 <= a\_r \times x\_r$!!

Sequential with IO limits



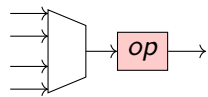
- 1st Cycle $\rightarrow a|b|c|x\_r <= a, b, c, x$
- 2nd Cycle $\rightarrow t1 <= a\_r \times x\_r$
- 3rd Cycle $\rightarrow t2 <= t1 + b\_r$
- 4th Cycle $\rightarrow t3 <= t2 \times x\_r$
- 5th Cycle $\rightarrow y <= t3 + c\_r,\ a|b|c|x\_r <= a, b, c, x$
- 6th Cycle $\rightarrow t1 <= a\_r \times x\_r$!!

Dataflow Analysis

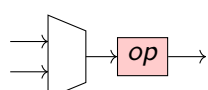
- Identify system constraints → how many resources we have?
  - Number and type of arithmetic operators
  - Number of IOs
  - Number of registers
- Inspect your problem to identify how many resources do you need for a **fully-spatial** implementation. (*i.e.* cost is not a concern)
- Put a **multiplexer** in front of **every** resource that is less than what you need.
  - Sharing is enabled with the multiplexer
  - In the simple case, assume **crossbar** connectivity → mux is connected to all possible inputs
- Extreme case is **fully-sequential** when you have exactly one of each resource → simple in-order CPU.

Hardware Design Goals

- Broadly, we must first design **correct** hardware, and then aim for **efficiency**
- The **implementation task** is to identify which operation is mapped to which resource in which cycle
  - Important to get this correct first
  - Think of this as writing an assembler for this custom datapath



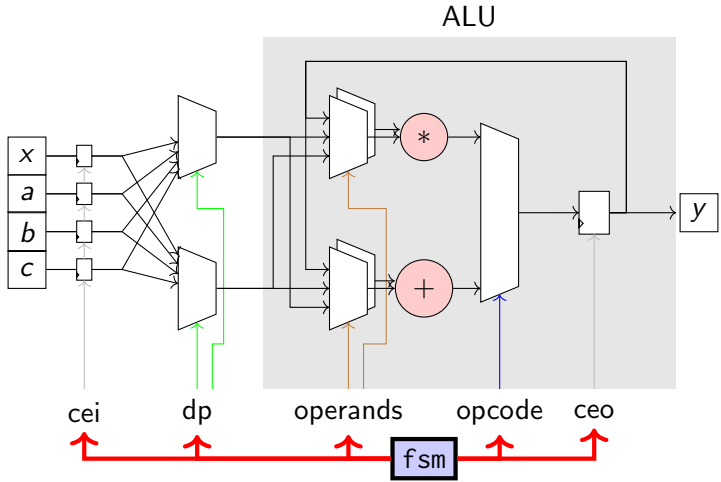
- The **optimization task** is to reduce the cost of multiplexers
  - We know crossbars scale as N^2 so can be expensive
  - On FPGAs multiplexers cost quite a bit. Mux cost often comparable to arithmetic units



Implementation Task

- First, we need to make a list of operations needed in the problem
  - op_1, op_2, ...
- For each resource, make a table of which operation is mapped to that resource in each cycle
  - Formally called a scheduling table
  - Like a time-table for a resource telling it what to do
  - This is then treated like a state-machine design problem
    - truth-table for logic implementation
- We need to alias (wraparound) the last cycle back to the 0th cycle to ensure full pipeline utilization.

Example poly walkthrough (bird's eye view)



Goal: Design a state machine to generate io, operands, and opcode signals.

Schedule Table for poly

```
poly:
  imull  %a, %x, %reg0
  addl   %reg0, %b, %reg0
  imull  %reg0, %x, %reg0
  addl   %reg0, %c, %reg0
  movl   %reg0, %y
  ret
```

Cycle	Operators	
	add_0	$mult_0$
0		
1		
2		
3		
4		

- Write down which operation is mapped to which arithmetic/logic resource in which cycle
  - Obey dependencies
  - Ensure that one resource only executes one operation in a cycle
- For now, this is a manual process → often sub-optimal when you have a choice *i.e.* multiple adders available
  - Use as few cycles as possible
  - Look for opportunities to reuse inputs across operations

Schedule Table for poly

```
poly:
  imull  %a, %x, %reg0
  addl   %reg0, %b, %reg0
  imull  %reg0, %x, %reg0
  addl   %reg0, %c, %reg0
  movl   %reg0, %y
  ret
```

Cycle	Operators	
	add_0	$mult_0$
0	–	–
1	–	$a\_r \cdot x\_r$
2	$reg_0 + b\_r$	–
3	–	$reg_0 \cdot x\_r$
4	$reg_0 + c\_r$	–

- Write down which operation is mapped to which arithmetic/logic resource in which cycle
  - Obey dependencies
  - Ensure that one resource only executes one operation in a cycle
- For now, this is a manual process → often sub-optimal when you have a choice *i.e.* multiple adders available
  - Use as few cycles as possible
  - Look for opportunities to reuse inputs across operations

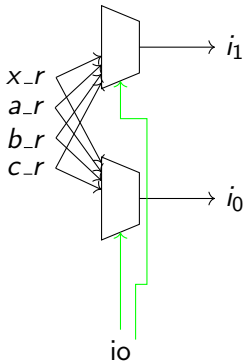
Schedule Table for poly

```
poly:
  imull  %a, %x, %reg0
  addl   %reg0, %b, %reg0
  imull  %reg0, %x, %reg0
  addl   %reg0, %c, %reg0
  movl   %reg0, %y
  ret
```

Cycle	Operators	
	add_0	$mult_0$
0	$reg_0 + c\_r$	–
1	–	$a\_r \cdot x\_r$
2	$reg_0 + b\_r$	–
3	–	$reg_0 \cdot x\_r$
4	$reg_0 + c\_r$	–

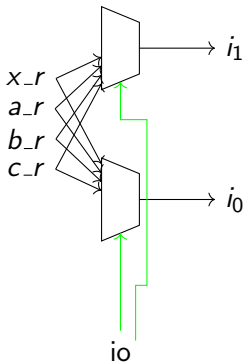
- Write down which operation is mapped to which arithmetic/logic resource in which cycle
  - Obey dependencies
  - Ensure that one resource only executes one operation in a cycle
- For now, this is a manual process → often sub-optimal when you have a choice *i.e.* multiple adders available
  - Use as few cycles as possible
  - Look for opportunities to reuse inputs across operations

Example poly walkthrough (Generate io)



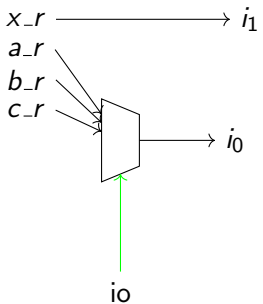
Cycle	Inputs	
	i_0	i_1
0		
1		
2		
3		
4		

Example poly walkthrough (Generate io)



Cycle	Inputs	
	i_0	i_1
0	—	—
1	a_r	x_r
2	b_r	—
3	—	x_r
4	c_r	—

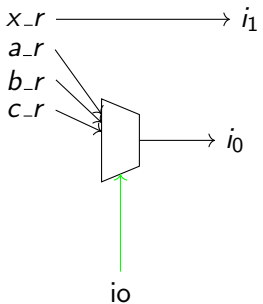
Example poly walkthrough (Generate io)



Cycle	Inputs	
	i_0	i_1
0	–	–
1	a_r	x_r
2	b_r	–
3	–	x_r
4	c_r	–

- i_1 does not need a mux, as we always select x
- i_0 never chooses x, so sufficient to use a 3-input mux.

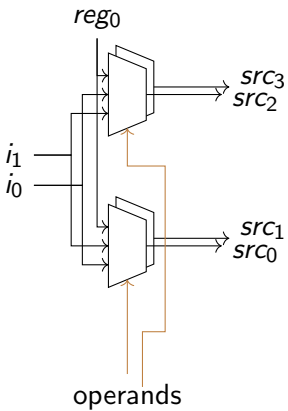
Example poly walkthrough (Generate io)



Cycle	Inputs	
	i_0	i_1
0	c_r	—
1	a_r	x_r
2	b_r	—
3	—	x_r
4	c_r	—

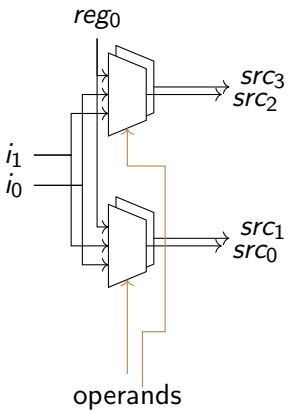
- i_1 does not need a mux, as we always select x
- i_0 never chooses x, so sufficient to use a 3-input mux.

Example poly walkthrough (Generate operands)



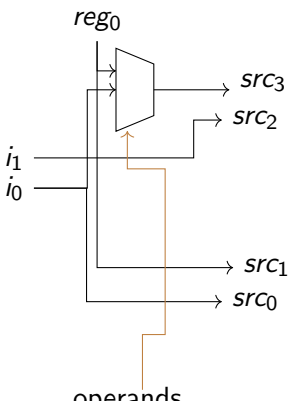
Cycle	add_0		$mult_0$	
	src_0	src_1	src_2	src_3
0				
1				
2				
3				
4				

Example poly walkthrough (Generate operands)



Cycle	add_0		$mult_0$	
	src_0	src_1	src_2	src_3
0	–	–	–	–
1	–	–	i_1	i_0
2	i_0	reg_0	–	–
3	–	–	i_1	reg_0
4	i_0	reg_0	–	–

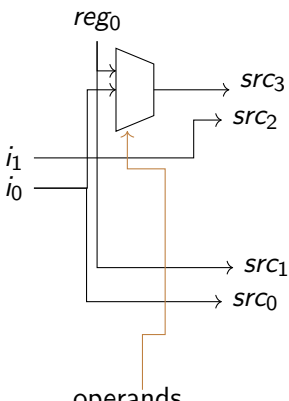
Example poly walkthrough (Generate operands)



Cycle	add_0		$mult_0$	
	src_0	src_1	src_2	src_3
0	–	–	–	–
1	–	–	i_1	i_0
2	i_0	reg_0	–	–
3	–	–	i_1	reg_0
4	i_0	reg_0	–	–

- src_0 and src_2 do not need muxes, as we always select i_0 and i_1 respectively.
- src_1 always uses the feedback edge reg_0. Again no mux needed.
- A simple 2:1 mux is required for src_3 to choose between i_0 and reg_0

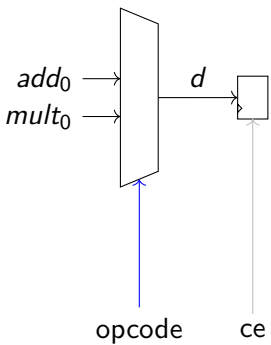
Example poly walkthrough (Generate operands)



Cycle	add_0		$mult_0$	
	src_0	src_1	src_2	src_3
0	i_0	reg_0	—	—
1	—	—	i_1	i_0
2	i_0	reg_0	—	—
3	—	—	i_1	reg_0
4	i_0	reg_0	—	—

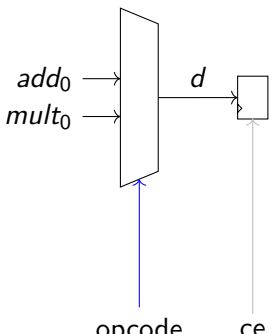
- src_0 and src_2 do not need muxes, as we always select i_0 and i_1 respectively.
- src_1 always uses the feedback edge reg_0. Again no mux needed.
- A simple 2:1 mux is required for src_3 to choose between i_0 and reg_0

Example poly walkthrough (Generate opcode)



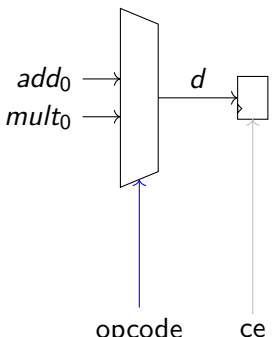
Cycle	reg_0	
	ce	*d*
0		
1		
2		
3		
4		

Example poly walkthrough (Generate opcode)



Cycle	reg_0	
	ce	*d*
0	–	–
1	1	$mult_0$
2	1	add_0
3	1	$mult_0$
4	1	add_0

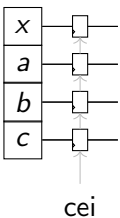
Example poly walkthrough (Generate opcode)



Cycle	reg_0	
	ce	d
0	1	add_0
1	1	$mult_0$
2	1	add_0
3	1	$mult_0$
4	1	add_0

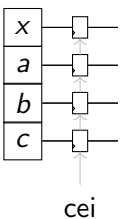
- ce is clock enable and its important to set this correctly to avoid overwriting the register with garbage
- d is the data input that must chosen from the set of possible operator outputs

Example poly walkthrough (Generate cei)



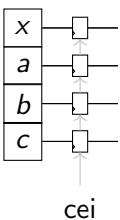
Cycle	*x_r*		*a_r*		*b_r*		*c_r*	
	ce	*d*	*ce*	*d*	*ce*	*d*	*ce*	*d*
0								
1								
2								
3								
4								

Example poly walkthrough (Generate cei)



Cycle	*x_r*		*a_r*		*b_r*		*c_r*	
	ce	*d*	*ce*	*d*	*ce*	*d*	*ce*	*d*
0	1	*x*	1	*a*	1	*b*	1	*c*
1	0	–	0	–	0	–	0	–
2	0	–	0	–	0	–	0	–
3	0	–	0	–	0	–	0	–
4	0	–	0	–	0	–	0	–

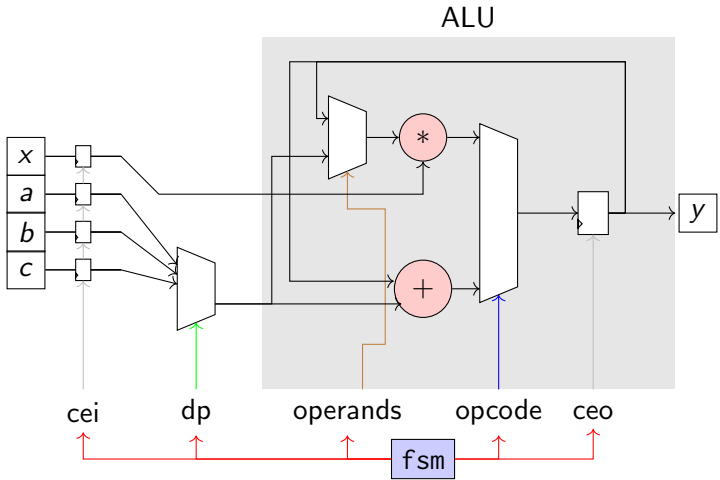
Example poly walkthrough (Generate cei)



Cycle	x_r		a_r		b_r		c_r	
	ce	*d*	*ce*	*d*	*ce*	*d*	*ce*	*d*
0	1	*x*	1	*a*	1	*b*	1	*c*
1	0	–	0	–	0	–	0	–
2	0	–	0	–	0	–	0	–
3	0	–	0	–	0	–	0	–
4	1	*x*	1	*a*	1	*b*	1	*c*

- For input register enables, we alias cycle 0 selections forward to the last cycle.
- *cei* is an input clock enable and only active for one cycle at the start.
- *d* is the data input and just wires the datapath input

After optimization poly

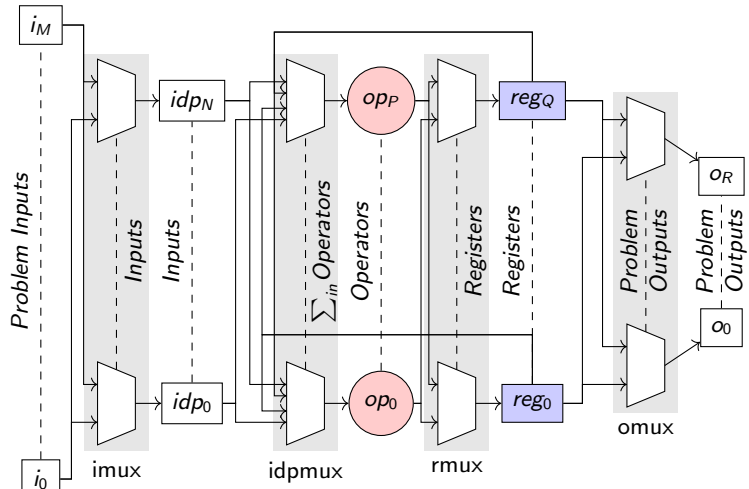


Verilog for optimized poly

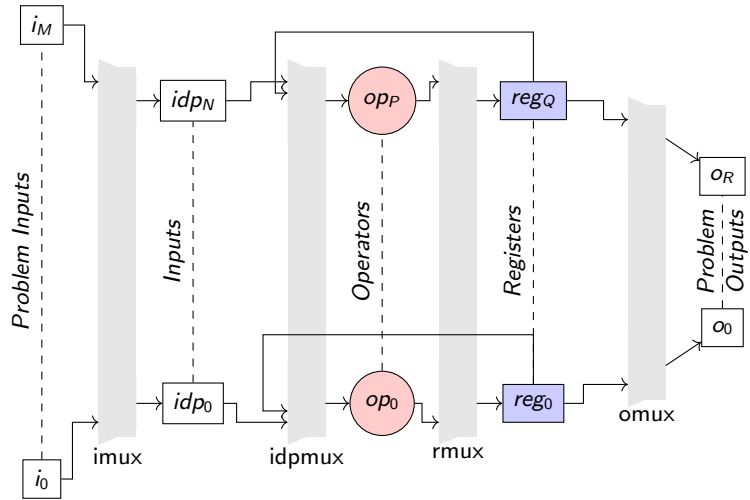
```
// generate fsm outputs explicitly
assign dp = count == 1 ? 2'b00 : count == 2 ? 2'b01 : 2'b10;
assign operands = count == 1 ? 1'b0 : 1'b1;
assign opcode = (count == 1 || count == 3) ? 1'b0 : 1'b1;
assign ceo = 1'b1;
assign cei = count == 0;
// imux
assign i0 = (dp == 2'b00) ? a_r : (dp == 2'b01) ? b_r : c_r;
assign i1 = x_r;
// dpmux
assign src0 = {16{1'b0},i0};
assign src1 = y_r;
assign src2 = i1;
assign src3 = (operands == 1'b0) ? {8{1'b0},i0} : y_r[15:0];
// arithmetic datapath
assign add0 = src0 + src1;
assign mult0 = src2 * src3;
```

```
always @(posedge clk) begin
 if(rst) begin
  y_r <= {24{1'b0}};
 end else begin
  if(ce) begin
   if(~opcode) begin
    y_r <= mult0;
   end else begin
    y_r <= add0;
   end
  end
 end
end
```

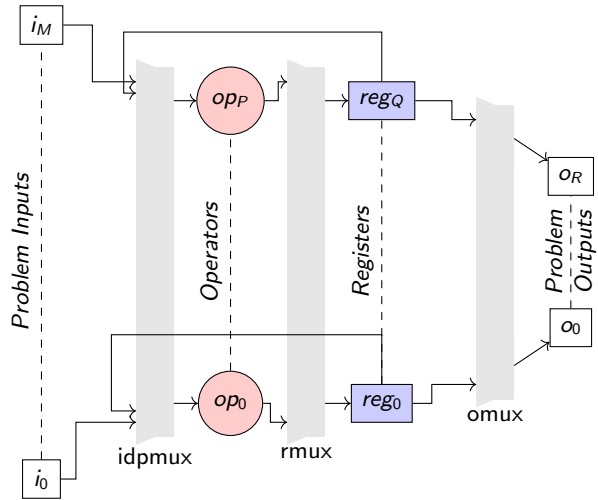
Datapath Design notation



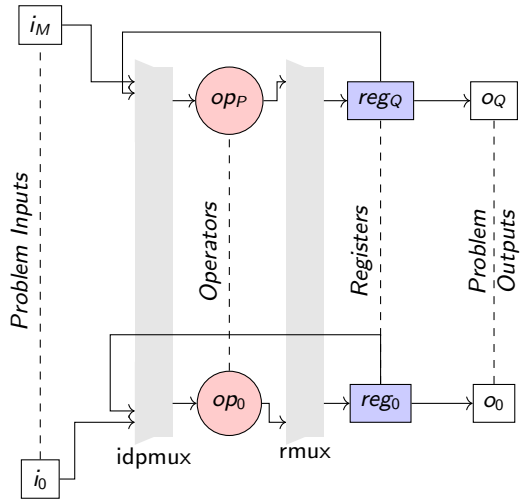
Datapath Design notation (Crossbar View)



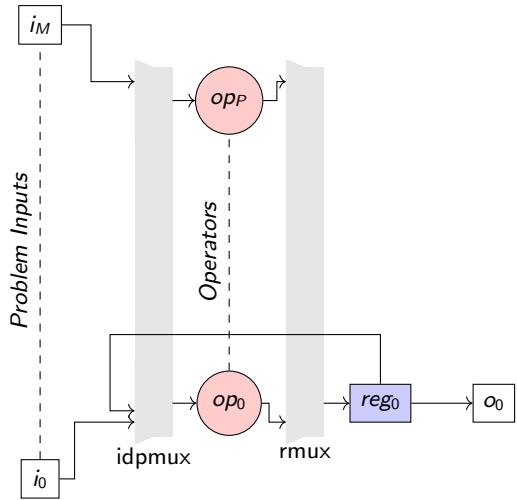
Datapath Design notation (No Datapath Input limits)



Datapath Design notation (No Datapath Input or Output limits)



Datapath Design notation (No Datapath Input Limits + Single Reg/Output)



Class Wrapup

- Resource sharing is a key design concept for hardware engineering
  - Sharing necessitated by limited chip capacity
  - Infrequent tasks should be allocated limited resources
- Static scheduling allows per-cycle control of hardware resources
- We are effectively doing a compilers job manually $\rightarrow$ HLS can automate this (after midterm)
- Optimization goal is to reduce the number of cycles required, cost of multiplexers, number of distinct operators