Memories

Nachiket Kapre
nachiket@uwaterloo.ca
UNIVERSITY OF WATERLOO

Outline

- Need for memory
  - CPU model (instruction + data)
- Kinds of Memories
  - Flip-Flops – Register File – SRAM – DRAM
- RTL Syntax for memories
- Timing diagrams for memories

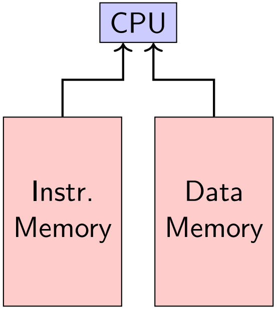
Need for memory

- **History**: For computability, you need to store history of events, or store intermediate results. Think of Turing Machine + infinite tape.
  - Cannot rely on environment to provide storage

Memory

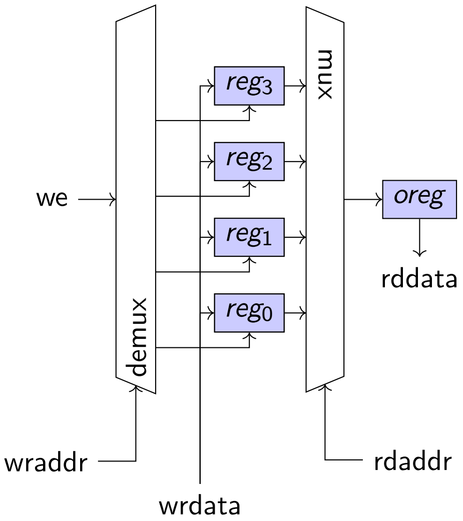
- **Cost**: When implementing storage in hardware, you have a choice of implementations. Must know when to use which kind of memory.
  - FFs are fast parallel access but expensive. Single-cycle access to all FFs or registers is possible.
    - Register Files are collections of FFs with multiple read/write ports
  - SRAMs (Static RAMs) are compact but only access one element at a time. Single-cycle access is possible.
  - DRAMs (Dynamic RAMs) are cheapest, but require several cycles per access
- **Scheduling**: Timing behavior of different memories is different. Must think of memory properties when scheduling a datapath.

Memory in CPUs



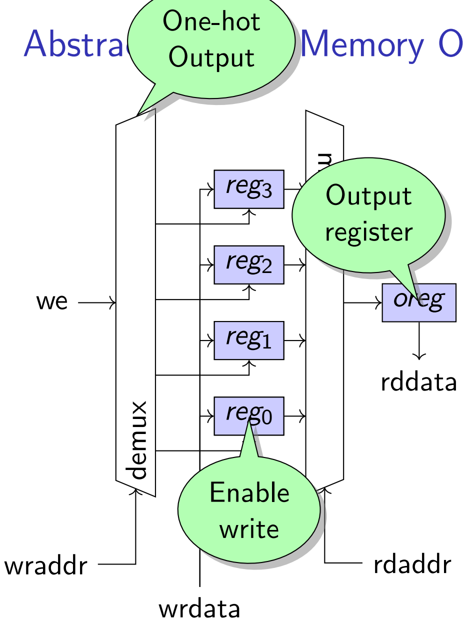
- Memory used to store instructions + data
- In hardware, instructions are just operations distributed in space + resource shared if required
- Variables on stack, dynamic allocation on heap → data memory
- Modern CPUs mix these two memory spaces, so we can write self-modifying code (downside: data can be made executable)
- For hardware design:
  - Focus on using `dmem` to store intermediate variables.
  - Implement `imem` for pre-compiled sequence of signals → microcode for datapath (state machine)

Abstract View of Memory Operations



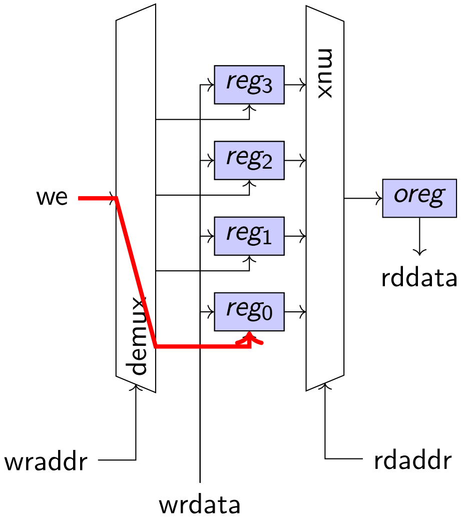
- Memory is a collection of storage locations (registers)
- **Mux** and **Demux** blocks arbitrate access the multiple memory locations
- Read address == select for the **Mux** block.
- Write address == select for the **Demux** block.
- Write enable = clock enable
- Reads are always happening, no enable needed

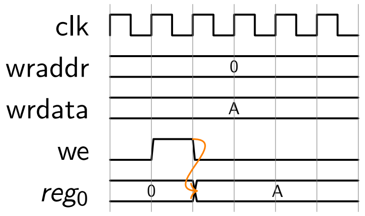
Abstra Memory Operations



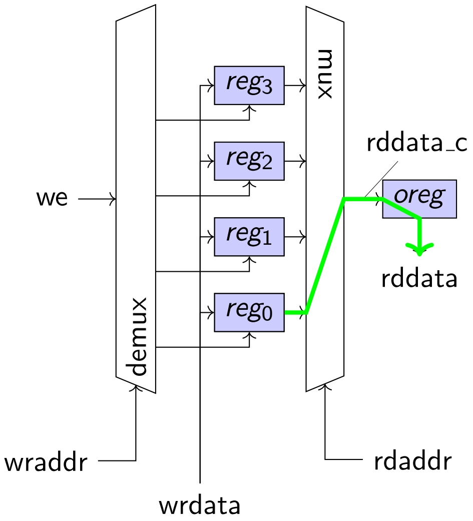
- Memory is a collection of storage locations (registers)
- **Mux** and **Demux** blocks arbitrate access the multiple memory locations
- Read address == select for the **Mux** block.
- Write address == select for the **Demux** block.
- Write enable = clock enable
- Reads are always happening, no enable needed

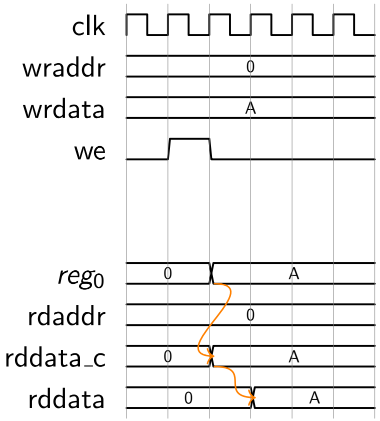
Abstract View of Memory Operations





Abstract View of Memory Operations





Memory behavior

- Reads and writes can happen in parallel
  - FPGA RAMs allow both ports to be read/write ports
- Reads are happening **all** the time on the **rdaddr** location. No explicit read enable is needed
- Writes need explicit write enable which becomes a **ce** input to register (memory location)
- Write data gets stored into register in one cycle
- Read data available one cycle after address provided, or register state changed
  - If **rdaddr** stable, output data changes one cycle after **reg**
  - If **reg** values are stable, output data changes on cycle after **rdaddr**
- Thus, end-to-end delay write+read to+from same location = 2 cycles

RTL Code for Memory Inference

```verilog
module mem #(
    parameter [31:0] ADDRWIDTH=8,
    parameter [31:0] DATAWIDTH=32) (
    input wire clk,
    input wire rst,
    input wire [DATAWIDTH - 1:0] wrdata,
    output reg [DATAWIDTH - 1:0] rddata,
    input wire [ADDRWIDTH - 1:0] wraddr,
    input wire [ADDRWIDTH - 1:0] rdaddr,
    input wire we
  );
  //
  reg [DATAWIDTH - 1:0] mem[2 ** ADDRWIDTH - 1:0];

  integer i;
  always @(posedge clk) begin
    if(rst) begin
      `ifndef SYNTHESIS //
        for (i=0; i <= 2**ADDRWIDTH - 1; i = i+1) begin
          mem[i] <= i;
        end
      `endif
      rddata <= 0;
    end else begin
      if(we) begin //
        mem[wraddr] <= wrdata;
      end
      rddata <= mem[rdaddr]; //
    end
  end
end
endmodule
```

- Define an array with two parameters
  - number of data bits
  - number of address bits
- Resetting RAM only OK for simulations
- Clocked always block to manage **simultaneous** read and write operations
- **Q**: What happens if read+write to same addr in same cyc?

RTL Code for Memory Inference

```
module mem #(
  parameter [31:0] ADDRWIDTH=8,
  parameter [31:0] DATAWIDTH=32) (
  input wire clk,
  input wire rst
  input wire [D          ] wrdata,
  output reg [           rddata,
  input wire [A          ddr,
  input wire
  input wir
 );
 //
 reg [DATAWI              H - 1:0];

 integer i;
 always @(posedge clk) b
  if(rst) begin
   `ifndef SYNTHESIS //
    for (i=0; i <=          i = i+1) begin
     mem[i] <=
    end
   `endif
   rddata <= 0;
  end else begin
   if(we) begin //
    mem[wraddr] <= wrdata;
   end
   rddata <= mem[rdaddr]; //
  end
 end
end
endmodule
```

2D

Initialization cannot be synthesized

Conditional Write

Unconditional Read

- Define an array with two parameters
  - number of data bits
  - number of address bits
- Resetting RAM only OK for simulations
- Clocked always block to manage **simultaneous** read and write operations
- **Q**: What happens if read+write to same addr in same cyc?

Using Memories

- On-chip memories can be used to store intermediate data, inputs, and outputs
- To access a memory, you need to supply read/write address + write control $\rightarrow$ state machine
- Data to/from memories typically goes to user circuit $\rightarrow$ resource-shared or fully-pipelined datapath
- Address pattern + controls must match the expected data consumption rate of the datapath
  - A fully-pipelined circuit can consume + produce a set of inputs+outputs per cycle $\rightarrow$ state machine is simply a counter supplying address to memory
  - A resource-shared datapath consumes inputs once every `THROUGHPUT` number of cycles $\rightarrow$ need explicit state machine

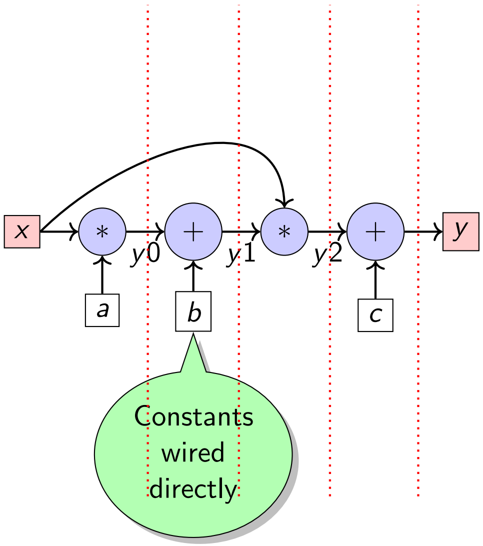
Using Memories with poly datapath (Fully-Pipelined)

- Compute the following function in hardware

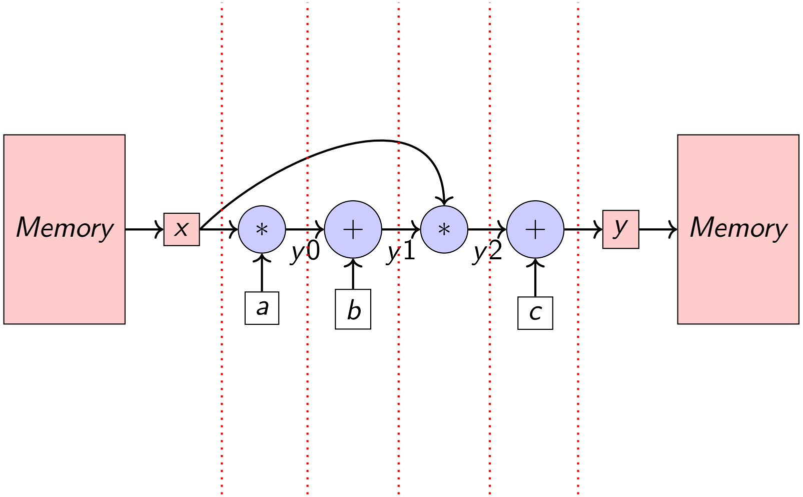
```
#define N 1024

volatile int x[N]={1,2,3,4};
volatile int y[N]={0,0,0,0};

void poly(int a, int b, int c)
{
 for(int i=0;i<N;i++) {
  y[i]=a*x[i]*x[i]+b*x[i]+c;
 }
}

int main() {
 poly(1,2,3);
}
```

- Assume a, b, and c are inputs
- Store x and y into memory blocks
- Need a state-machine/counter to loop over items of x and y

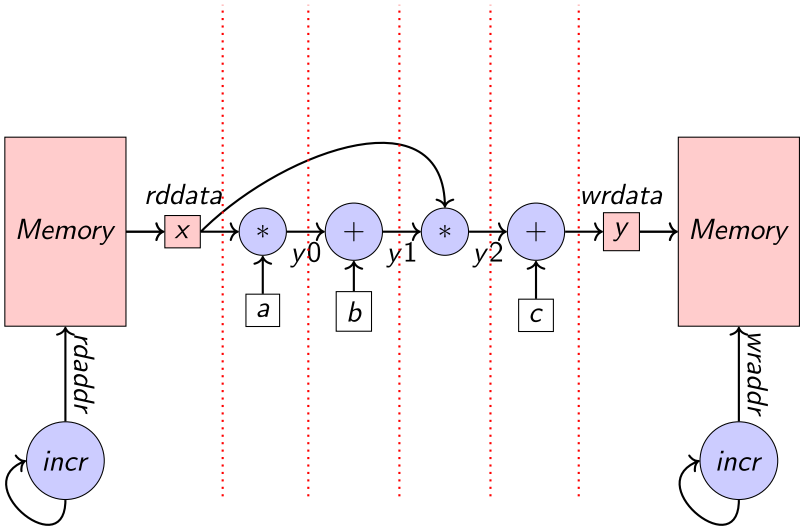
Using Memories with `poly` datapath (Fully-Pipelined)



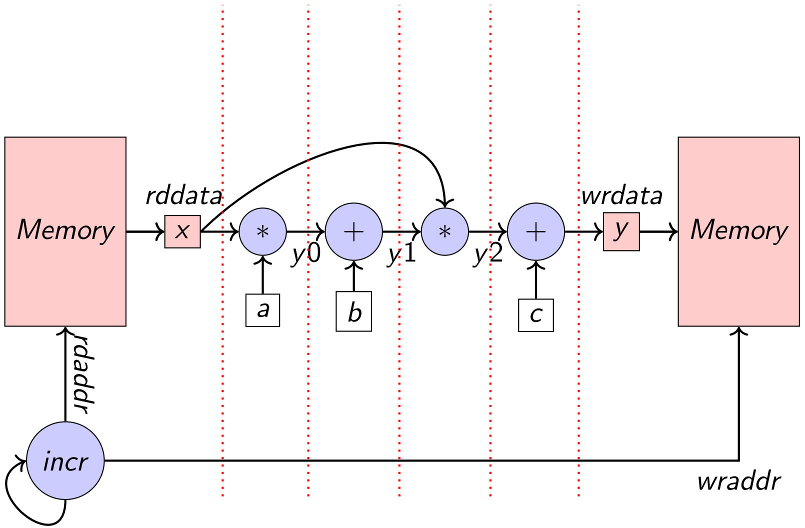
Using Memories with `poly` datapath (Fully-Pipelined)



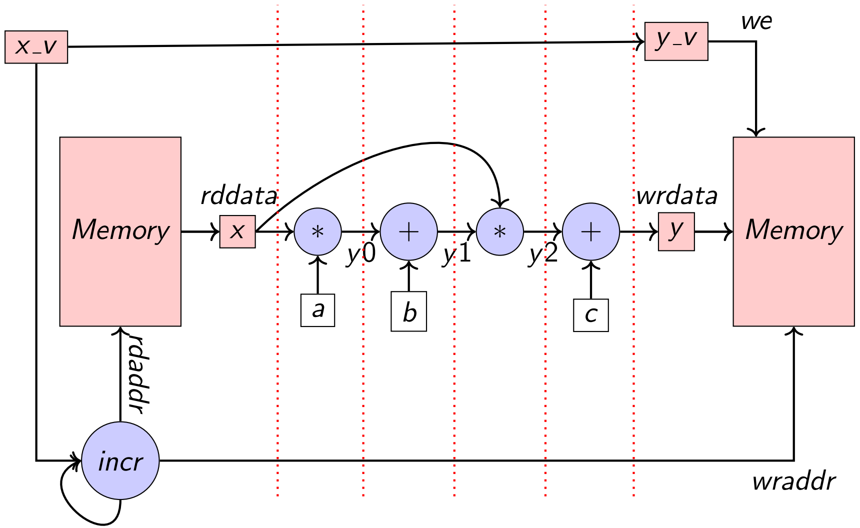
Using Memories with `poly` datapath (Fully-Pipelined)



Using Memories with poly datapath (Fully-Pipelined)



Using Memories with poly datapath (Fully-Pipelined)



Verilog Code for poly + Memory + Fully-Pipelined

```
poly_dp #(
 .DATAWIDTH(DATAWIDTH))
dp_inst (
 .clk(clk),
 .rst(rst),
 .a(a),
 .b(b),
 .c(c),
 .x(x),
 .y(y));

poly_stmc #(
 .ADDRWIDTH(ADDRWIDTH))
stmc_inst(
 .clk(clk),
 .rst(rst),
 .valid(x_v),
 .rdaddr(rdaddr),
 .wraddr(wraddr),
 .we(we));
```

```
mem #(
 .ADDRWIDTH(ADDRWIDTH),
 .DATAWIDTH(DATAWIDTH))
x_mem_inst(
 .clk(clk),
 .rst(rst),
 .rddata(x),
 .wrdata({DATAWIDTH{1'b0}}),
 .wraddr({ADDRWIDTH{1'b0}}),
 .rdaddr(rdaddr),
 .we(1'b 0));

mem #(
 .ADDRWIDTH(ADDRWIDTH),
 .DATAWIDTH(3*DATAWIDTH))
y_mem_inst(
 .clk(clk),
 .rst(rst),
 .wrdata(y),
 .wraddr(wraddr),
 .rdaddr({ADDRWIDTH{1'b0}}),
 .rddata(),
 .we(we));

  assign y_v = we;
```

Verilog Code for State Machine

```
  if(rst) begin
   valid_r <= 1'b 0; valid_r1 <= 1'b 0;
   valid_r2 <= 1'b 0; valid_r3 <= 1'b 0;
   valid_r4 <= 1'b 0;
   addr_r <= {ADDRWIDTH{1'b0}};
   addr_r1 <= {ADDRWIDTH{1'b0}};
   addr_r2 <= {ADDRWIDTH{1'b0}};
   addr_r3 <= {ADDRWIDTH{1'b0}};
   addr_r4 <= {ADDRWIDTH{1'b0}};
   addr_r5 <= {ADDRWIDTH{1'b0}};
  end else begin
   valid_r <= valid;
   valid_r1 <= valid_r;
   valid_r2 <= valid_r1; //
   valid_r3 <= valid_r2;
   valid_r4 <= valid_r3;
   if(valid) begin //
    addr_r <= addr_r + 1;
   end
   addr_r1 <= addr_r;
   addr_r2 <= addr_r1;
   addr_r3 <= addr_r2;
   addr_r4 <= addr_r3; //
   addr_r5 <= addr_r4;
  end
 end

 assign rdaddr = addr_r;
 //
 assign wraddr = addr_r5;
 assign we = valid_r4;
```

- State machine generates addresses + write enable
- valid is input from external world
- rdaddr generated from a simple incrementer
  - incrementer controlled by valid
- wraddr and we are simply delayed versions of rdaddr and valid
  - Handled exactly like bubbles
- Notice that wraddr and we are offset by a cycle

Verilog Code for State Machine

```
    if(rst) begin
     valid_r <= 1'b 0; valid_r1 <= 1'b 0;
     valid_r2 <= 1'b 0; valid_r3 <= 1'b 0;
     valid_r4 <= 1'b 0;
     addr_r <= {ADDRWIDTH{1'b0}};
     addr_r1 <= {ADDRWIDTH{1'b0}};
     addr_r2 <= {ADDRWIDTH{1'b0}};
     addr_r3 <= {ADDRWIDTH{1'b0}}
     addr_r4 <= {ADDRWIDTH{1'b
     addr_r5 <= {ADDRWIDTH{1'
    end else begin
     valid_r <= valid;
     valid_r1 <= valid_r;
     valid_r2 <= valid_r1; //
     valid_r3 <= valid_r2;
     valid_r4 <= valid_r3;
     if(valid) begin //
      addr_r <= addr_r + 1;
     end
     addr_r1 <= addr_r;
     addr_r2 <= addr_r1;
     addr_r3 <= addr_r2;
     addr_r4 <= addr_r3; //
     addr_r5 <= addr_r4;
    end
   end

   assign rdaddr = addr_r;
   //
   assign wraddr = addr_r5;
   assign we = valid_r4;
```

valid pipeline

rdaddr generation

wraddr pipeline

- State machine generates addresses + write enable
- valid is input from external world
- rdaddr generated from a simple incrementer
  - incrementer controlled by valid
- wraddr and we are simply delayed versions of rdaddr and valid
  - Handled exactly like bubbles
- Notice that wraddr and we are offset by a cycle

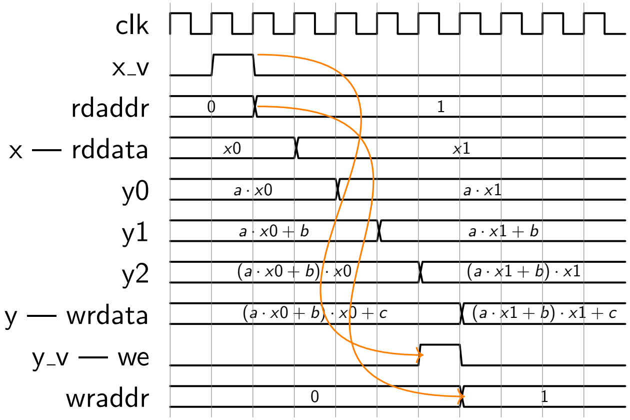
Verilog Code for State Machine

```
  if(rst) begin
   valid_r <= 1'b 0; valid_r1 <= 1'b 0;
   valid_r2 <= 1'b 0; valid_r3 <= 1'b 0;
   valid_r4 <= 1'b 0;
   addr_r <= {ADDRWIDTH{1'b0}};
   addr_r1 <= {ADDRWIDTH{1'b0}};
   addr_r2 <= {ADDRWIDTH{1'b0}};
   addr_r3 <= {ADDRWIDTH{1'b0}};
   addr_r4 <= {ADDRWIDTH{1'b0}};
   addr_r5 <= {ADDRWIDTH{1'b0}};
  end else begin
   valid_r <= valid;
   valid_r1 <= valid_r;
   valid_r2 <= valid_r1; //
   valid_r3 <= valid_r2;
   valid_r4 <= valid_r3;
   if(valid) begin //
    addr_r <= addr_r + 1;
   end
   addr_r1 <= addr_r;
   addr_r2 <= addr_r1;
   addr_r3 <= addr_r2;
   a                 //


en

assign rdaddr = a   r_r;
//
assign wraddr = addr_r5;
assign we = valid_r4;
```

One cycle offset

- State machine generates addresses + write enable
- valid is input from external world
- rdaddr generated from a simple incrementer
  - incrementer controlled by valid
- wraddr and we are simply delayed versions of rdaddr and valid
  - Handled exactly like bubbles
- Notice that wraddr and we are offset by a cycle

Timing Diagram of poly datapath



Sharing Memories

- In the general case, consider memory ports as resources
- Track read/write operations on each memory port and disallow multiple reads/writes to the memory
  - Multi-ported memories are possible, but they are expensive, and should be used in high-performance designs only
- Generate address + write enables based on table

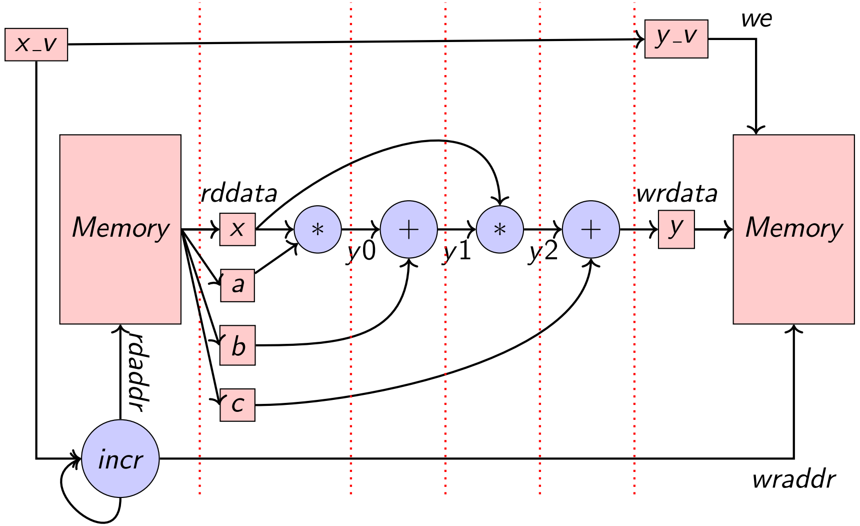
Using Shared Memories with poly datapath

- Compute the following function in hardware

```
#define N 1024

volatile int a[N]={1,2,3,4};
volatile int b[N]={1,2,3,4};
volatile int c[N]={1,2,3,4};
volatile int x[N]={1,2,3,4};
volatile int y[N]={0,0,0,0};

void poly()
{
 for(int i=0;i<N;i++) {
  y[i]=a[i]*x[i]*x[i]+b[i]*x[i]+c[i];
 }
}

int main() {
 poly();
}
```

- Now, a, b, c and x are all inputs per iteration
- Cannot afford distinct memories for each array
- How do we resource-share the memory port across these arrays?
- Need a state-machine/counter to manage things properly

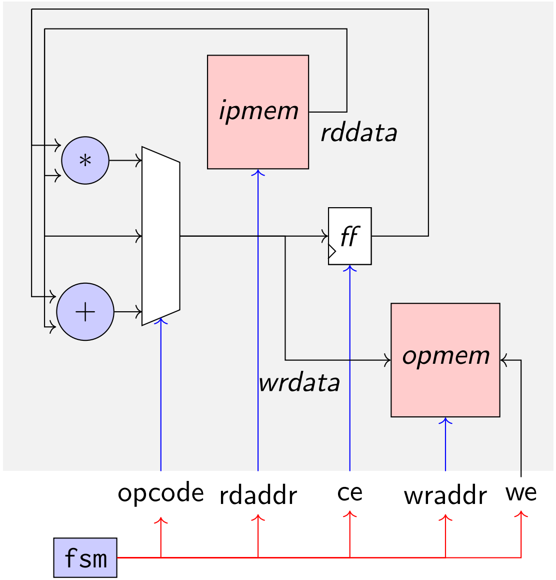
Using Shared Memories with `poly` (Fully-Pipelined)



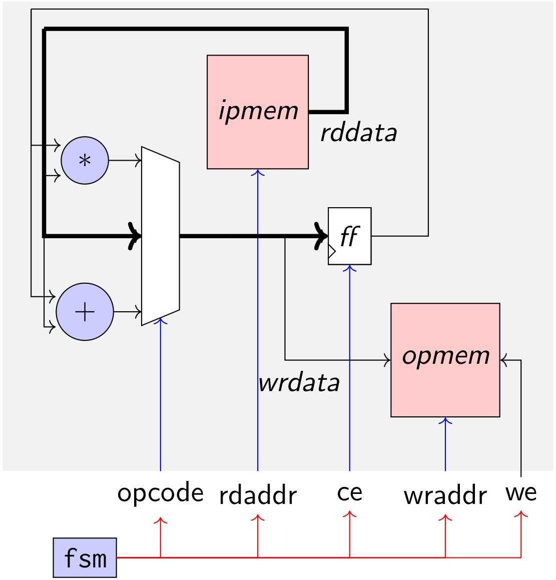
Limits of using Shared Memories with Fully-Pipelined Datapaths

- Shared memories require multiple cycles to fetch/store inputs, outputs, or temporary variables in the computation
- A fully-pipelined/fully-spatial circuit would waste resources while accessing data from memory
- Resources will stay idle waiting for memory operations
- Engineering optimization $\rightarrow$ find system bottlenecks, reallocate resources to target bottleneck
- Can we reduce resource cost? $\rightarrow$ resource-shared datapath!

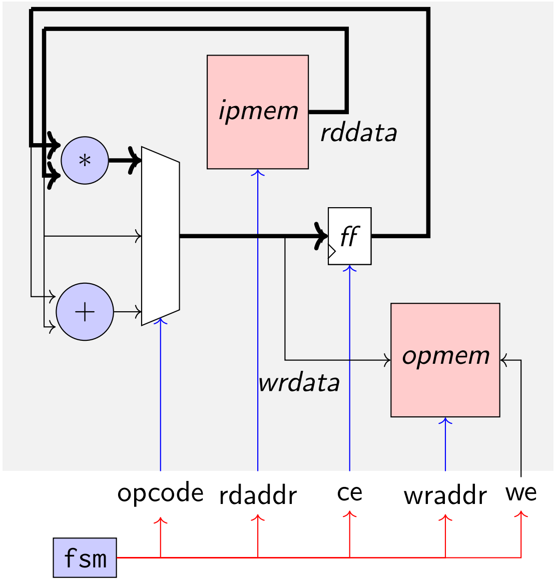
Using Shared Memories with poly (Resource-Shared)



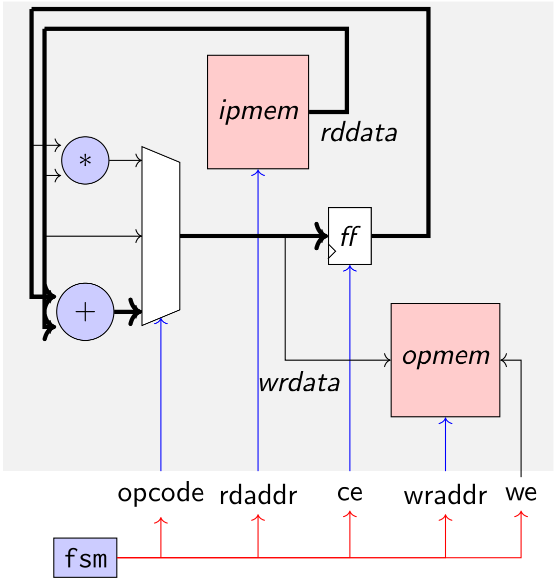
Using Shared Memories with poly (Resource-Shared)



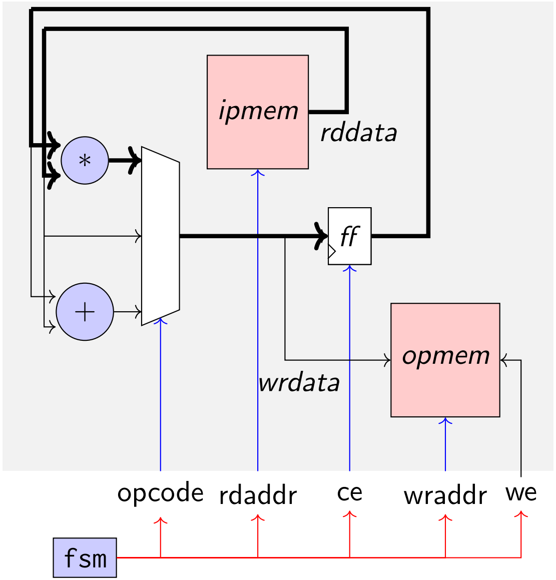
Using Shared Memories with poly (Resource-Shared)



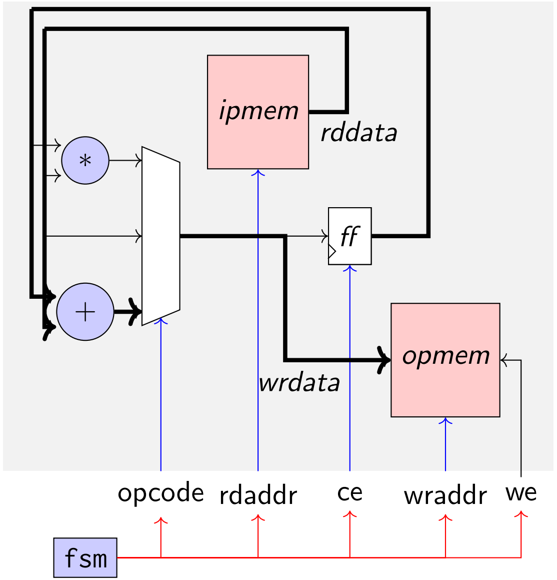
Using Shared Memories with poly (Resource-Shared)



Using Shared Memories with `poly` (Resource-Shared)



Using Shared Memories with `poly` (Resource-Shared)



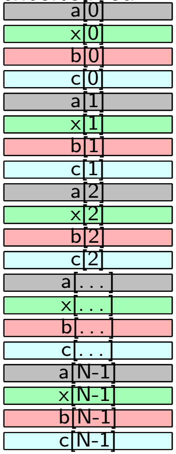
Revisiting Tables

Cycle	**Operators**			
	add_0	$mult_0$	$load_0$	$store_0$
0	–	–	a	–
1	–	$reg_0 \cdot x$	x	–
2	$reg_0 + b$	–	b	–
3	–	$reg_0 \cdot x$	x	–
4	$reg_0 + c$	–	c	$y = add_0$

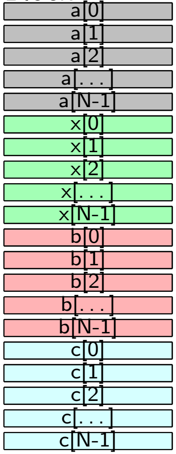
- Pack all input arrays x, a, b, and c into *ipmem*. Output y array stored in *opmem*.
- Modify schedule table to load a input in first cycle
- Modify datapath to provide a third “load” input to mux
- Note that no multiplexers required as input to operators $\rightarrow$ all inputs in same memory
  - If multiple input memories used, muxes need to be put back

Memory Layout

Interleaved



Block



Address Generation (Interleaved)

Cycle	**Memory Interface**		
	rdaddr	*wraddr*	*we*
0	0 ($a0$)	–	0
1	1 ($x0$)	–	0
2	2 ($b0$)	–	0
3	1 ($x0$)	–	0
4	3 ($c0$)	0 ($y0$)	1

- Mux selection replaced by address generation
- Address to memory   muxsel to a set of memory location
- Data layout is an important concern
  - Interleaved a0,x0,b0,c0,a1,x1,. . .
  - Block a0,a1,. . . ,x0,x1,. . .
  - Interleaved order simplifies address generation logic for this case
- Interleaving produces a simple state machine for generating read/write addresses.

C-like code equivalence

```
#define N 1024

volatile int abcx[4*N]={1,2,3,4};
volatile int y[N]={0,0,0,0};

void poly()
{
 for(int i=0;i<N;i++) {
  int a = abcx[4*i+0];
  int x = abcx[4*i+1];
  int b = abcx[4*i+2];
  int c = abcx[4*i+3];
  y[i]=a*x*x+b*x+c;
 }
}

int main() {
 poly();
}
```

- Combine a, b, c, and x arrays into a single array abcx $\rightarrow$ sort-of mimic imem in hardware
- Assume interleaved packing a0,x0,b0,c0,a1,x1,b1,c1,...
- Must compute index into abcx as a function of loop index *i*

State Machine for Resource-Shared FSM

```
  if(rst) begin
   counter <= {3{1'b0}};
   i <= {ADDRWIDTH{1'b0}};
  end else begin
   counter <= counter + 1;
   if((counter == (TABLE_SIZE - 1))) begin
    counter <= {ADDRWIDTH{1'b0}};
    i <= i + 1;
   end
  end
 end
```

```
case(counter)
 0 : begin
  rdaddr <= TABLE_SIZE * i + 0;
  wraddr <= {ADDRWIDTH{1'b0}};
  we <= 1'b 0;
 end
 1 : begin
  rdaddr <= TABLE_SIZE * i + 1;
  wraddr <= {ADDRWIDTH{1'b0}};
  we <= 1'b 0;
 end
 2 : begin
  rdaddr <= TABLE_SIZE * i + 2;
  wraddr <= {ADDRWIDTH{1'b0}};
  we <= 1'b 0;
 end
 3 : begin
  rdaddr <= TABLE_SIZE * i + 1;
  wraddr <= {ADDRWIDTH{1'b0}};
  we <= 1'b 0;
 end
 4 : begin
  rdaddr <= TABLE_SIZE * i + 3;
  wraddr <= i;
  we <= 1'b 1;
 end
 default : begin
  rdaddr <= TABLE_SIZE * i + 0;
  wraddr <= {ADDRWIDTH{1'b0}};
  we <= 1'b 0;
 end
endcase
```

Benefits of Resource Shared Datapath + Shared Memories

- Carefully balance resources against memory access bottleneck
- Isolated from rest of chip through `ipmem` and `opmem` structures
- Input and datapath muxes vanish $\rightarrow$ folded into the memory address decoders (internal to RAM structures)
- Now, we must create scheduling tables for memory read/write ports just like we did for operators and registers

Wrapup

- Memories are efficient ways to store large number of registers
- Various kinds of memories on offer – multi-ported register files, SRAMs, DRAMs
- Full-throughput designs use exclusive memories for all arrays, no conflicts
- Resource-shared memories better match for resource-shared datapaths $\rightarrow$ area-time tradeoff