

Advanced Memories

Nachiket Kapre

nachiket@uwaterloo.ca



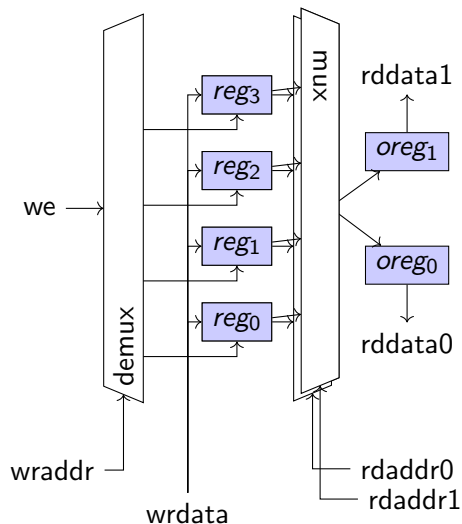
Outline

- ▶ Multi-ported memories
- ▶ Shift Registers
- ▶ FIFOs
- ▶ Loop Unrolling

Multi-ported Memories

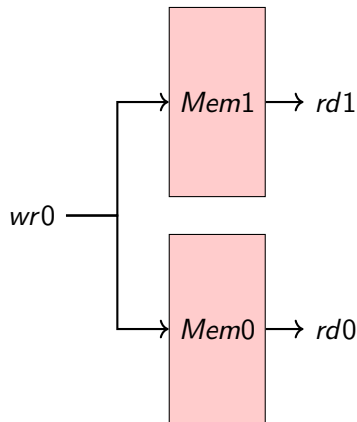
- ▶ Modern CPU register files allow multiple reads/writes
- ▶ Even simple instruction like $y = a + b$ require two reads + one write
- ▶ For superscalar multiple-issue processors, you could need several ports
- ▶ How do we provide multiple ports?
- ▶ How do we handle hazards (multiple writes to same location)?

Multiple read ports to register file



- ▶ Recall, that memory is just a collection of registers stored compactly
- ▶ We can connect register outputs to two **Mux** instantiations → load on each registers is doubled
- ▶ Unfortunately impractical for larger memories → multiple wires per Q port of the register, slower wires tend to be more expensive

Practical Multiple Read Port RAM



- ▶ Providing two read ports is easy → duplicate the RAM
- ▶ Only one write port available that repeats the write
- ▶ Multi-ported RAMs used in practice
 - ▶ “2.3GHz Wire-Speed POWER processor replicates a 2-read SRAM bank to achieve 4 read ports”
 - ▶ “Alpha 21264 microprocessor has a replicated 80-entry register file, thus doubling the number of read ports to support two concurrent integer execution units”

Multiple Read Port RAM Verilog

```
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
        rddata0 <= 0;
        rddata1 <= 0;
    end else begin
        if(we) begin //
            mem[wraddr] <= wrdata;
        end
        rddata0 <= mem[rdaddr0];
        rddata1 <= mem[rdaddr1];
        //
    end
end

endmodule
```

Multiple Read Port RAM Verilog

```
reg [DATAWIDTH - 1:0] mem[2 ** ADDRWIDTH - 1:0];

integer i;
always @(posedge clk) begin
    if(rst) begin
`ifdef SYNTHESIS
        for (i=0; i<=2**ADDRWIDTH-1; i=i+1) begin
            mem[i] <= i;
        end
    `endif
    rd
    rdd
end else
    if(we) begin
        mem[wraddr] <= wrdata;
    end
    rddata0 <= mem[rdaddr0];
    rddata1 <= mem[rdaddr1];
    //
end
end

endmodule
```

Single write
same RAM

Two reads
same RAM

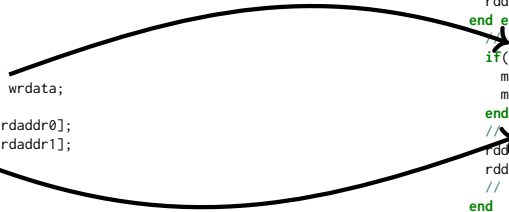
Multiple Read Port RAM Verilog

```
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
    rddata0 <= 0;
    rddata1 <= 0;
    end else begin
        if(we) begin //
            mem[wraddr] <= wrdata;
        end
        rddata0 <= mem[rdaddr0];
        rddata1 <= mem[rdaddr1];
        //
    end
end
endmodule
```

```
reg [DATAWIDTH - 1:0] mem0[2 ** ADDRWIDTH - 1:0];
reg [DATAWIDTH - 1:0] mem1[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
            mem0[i] <= i;
            mem1[i] <= i;
        end
    `endif
    rddata0 <= 0;
    rddata1 <= 0;
    end else begin
        if(we) begin //
            mem0[wraddr] <= wrdata;
            mem1[wraddr] <= wrdata;
        end
        //
        rddata0 <= mem0[rdaddr0];
        rddata1 <= mem1[rdaddr1];
        //
    end
end
endmodule
```



The diagram illustrates the transformation of a single-port RAM into a dual-port RAM. Two curved arrows originate from the left code block and point to the right code block. The first arrow starts at the line `mem[wraddr] <= wrdata;` and points to `mem0[wraddr] <= wrdata;`. The second arrow starts at the line `rddata0 <= mem[rdaddr0];` and points to `rddata0 <= mem0[rdaddr0];`. This shows how the single memory array is replaced by two separate arrays, `mem0` and `mem1`, to support multiple read ports.

Multiple Read Port RAM Verilog

```
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
    rddata0 <= 0;
    rddata1 <= 0;
    end else begin
        if(we) begin //
            mem[wraddr] <= wrdata;
        end
        rddata0 <= mem[rdaddr0];
        rddata1 <= mem[rdaddr1];
        //
    end
end
endmodule
```

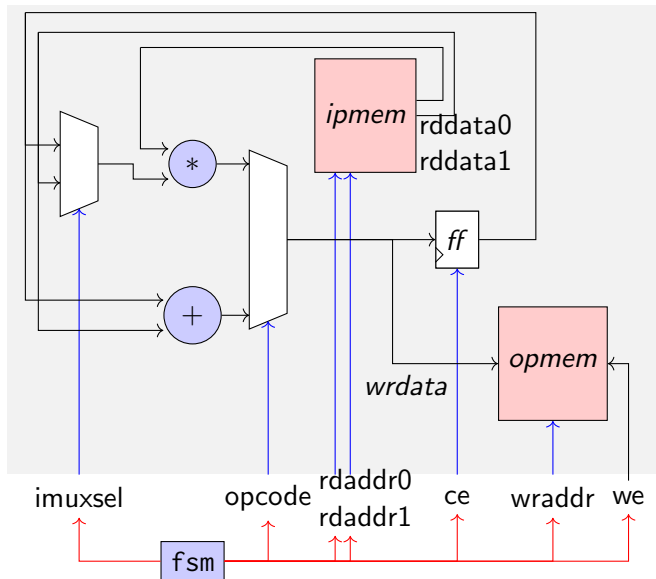
```
reg [DATAWIDTH - 1:0] mem0[2 ** ADDRWIDTH - 1:0];
reg [DATAWIDTH - 1:0] mem1[2 ** ADDRWIDTH - 1:0];

integer i;
always @(posedge clk) begin
    if(rst) begin
`ifndef SYNTHESIS
        for (i=0; i<=2**ADDRWIDTH-1; i=i+1) begin
            mem0[i] <= i;
            mem1[i] <= i;
        end
    `endif
    end else begin
        if(we) begin //
            mem0[wraddr] <= wrdata;
            mem1[wraddr] <= wrdata;
        end
        //
        rddata0 <= mem0[rdaddr0];
        rddata1 <= mem1[rdaddr1];
        //
    end
end
end
```

Parallel write
two RAMs

Parallel reads
two RAMs

Using Shared Memories with poly (Resource-Shared)



Schedule Table with Dual Read Port RAM

Cycle	Operators				
	add_0	$mult_0$	$load_0$	$load_1$	$store_0$
0	—	—	—	—	—
1	—	$a \cdot x$	a	x	—
2	$reg_0 + b$	—	b	—	—
3	—	$reg_0 \cdot x$	x	—	—
4	$reg_0 + c$	—	c	—	$y = add_0$

- ▶ Now, we need $load_0$ and $load_1$ ports for supporting two reads per cycle
- ▶ Two operands can be directly read from the memory in the same cycle
- ▶ Memory is registered, hence cycle 0 has no work.

Schedule Table with Dual Read Port RAM

Cycle	Operators				
	add_0	$mult_0$	$load_0$	$load_1$	$store_0$
0	$reg_0 + c$	—	c	—	$y = add_0$
1	—	$a \cdot x$	a	x	—
2	$reg_0 + b$	—	b	—	—
3	—	$reg_0 \cdot x$	x	—	—
4	$reg_0 + c$	—	c	—	$y = add_0$

- ▶ Now, we need $load_0$ and $load_1$ ports for supporting two reads per cycle
- ▶ Two operands can be directly read from the memory in the same cycle
- ▶ Memory is registered, hence cycle 0 has no work.

Address Generation (Interleaved)

Cycle	Memory Interface			
	<i>rdaddr0</i>	<i>rdaddr1</i>	<i>wraddr</i>	<i>we</i>
0	1 (<i>x0</i>)	0 (<i>a0</i>)	—	0
1	2 (<i>b0</i>)	—	—	0
2	1 (<i>x0</i>)	—	—	0
3	3 (<i>c0</i>)	—	—	0
4	—	—	0 (<i>y0</i>)	1

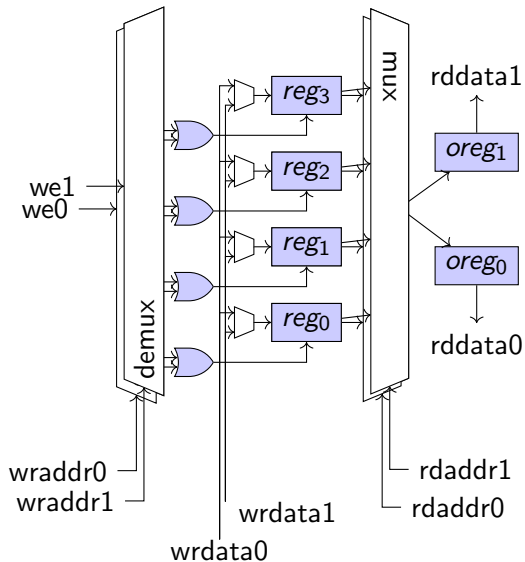
- ▶ Now we must create a column for each *rdaddr*
- ▶ Second memory port effectively underutilized, accessed only once every 4 cycles

Address Generation (Interleaved)

Cycle	Memory Interface			
	<i>rdaddr0</i>	<i>rdaddr1</i>	<i>wraddr</i>	<i>we</i>
0	1 (<i>x0</i>)	0 (<i>a0</i>)	—	0
1	2 (<i>b0</i>)	—	—	0
2	1 (<i>x0</i>)	—	—	0
3	3 (<i>c0</i>)	—	—	0
4	1 (<i>x1</i>)	0 (<i>a1</i>)	0 (<i>y0</i>)	1

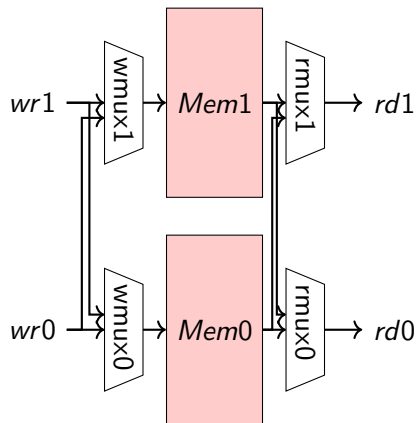
- ▶ Now we must create a column for each *rdaddr*
- ▶ Second memory port effectively underutilized, accessed only once every 4 cycles

Multiple write ports to register file



- ▶ Write enables must be decoded independently to allow two writes to happen at same time → demux + or
- ▶ Write data must be multiplexed at each register location → mux
- ▶ Expensive to support concurrent writes

Practical Multiple Write RAM



- ▶ Providing two write ports can be done with banking
 - ▶ **Condition:** Cannot do multiple writes to same bank, Cannot do multiple reads from same bank
 - ▶ Banking partitions memory of size M into k banks of size $\frac{M}{k}$
 - ▶ It IS NOT replication → replication creates two memories of size M each

Multiple Write Port RAM Verilog (Assuming no conflicts)

```
assign wraddr_b0 = (we0 & !wraddr0[ADDRWIDTH-1])? wraddr0[ADDRWIDTH-2:0]: wraddr1[ADDRWIDTH-2:0];
assign wrdata_b0 = (we0 & !wraddr0[ADDRWIDTH-1])? wrdata0[ADDRWIDTH-2:0]: wrdata1[ADDRWIDTH-2:0];
assign we_b0 = (we0 & !wraddr0[ADDRWIDTH-1]) | (we1 & !wraddr1[ADDRWIDTH-1]);
assign wraddr_b1 = (we1 & wraddr1[ADDRWIDTH-1])? wraddr1[ADDRWIDTH-2:0]: wraddr0[ADDRWIDTH-2:0];
assign wrdata_b1 = (we1 & wraddr1[ADDRWIDTH-1])? wrdata1[ADDRWIDTH-2:0]: wrdata0[ADDRWIDTH-2:0];
assign we_b1 = (we0 & wraddr0[ADDRWIDTH-1]) | (we1 & wraddr1[ADDRWIDTH-1]);

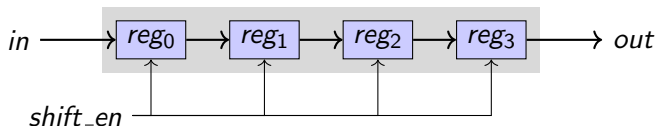
assign rdaddr_b0 = !rdaddr0[ADDRWIDTH-1]?rdaddr0[ADDRWIDTH-2:0]:rdaddr1[ADDRWIDTH-2:0];
assign rdaddr_b1 = rdaddr1[ADDRWIDTH-1]?rdaddr1[ADDRWIDTH-2:0]:rdaddr0[ADDRWIDTH-2:0];
assign rddata0 = !rdaddr0_r[ADDRWIDTH-1]?rddata_b0:rddata_b1;
assign rddata1 = rdaddr1_r[ADDRWIDTH-1]?rddata_b1:rddata_b0;
```

Multiple Write Port RAM Verilog (Assuming no conflicts)

```
always @(posedge clk) begin
  if(rst) begin
    `ifndef SYNTHESIS
      for (i=0; i <= 2**ADDRWIDTH - 2; i = i+1) begin
        mem0[i] <= i;
        mem1[i] <= i;
      end
    `endif
    rddata_b0 <= 0;
    rddata_b1 <= 0;
  end else begin
    if(we_b0) begin
      mem0[waddr_b0[ADDRWIDTH-2:0]] <= wrdata_b0;
    end
    if(we_b1) begin
      mem1[waddr_b1[ADDRWIDTH-2:0]] <= wrdata_b1;
    end
    rddata_b0 <= mem0[rdaddr_b0[ADDRWIDTH-2:0]];
    rddata_b1 <= mem1[rdaddr_b1[ADDRWIDTH-2:0]];
  end
end

always @(posedge clk) begin
  if(rst) begin
    rdaddr0_r <= 0;
    rdaddr1_r <= 0;
  end else begin
    rdaddr0_r <= rdaddr0;
    rdaddr1_r <= rdaddr1;
  end
end
```

Shift Registers



- ▶ A shift register is a fixed-sized memory that adds storage + latency to a signal
- ▶ Constant (minimum) latency for data passing through a Shift Register
- ▶ Popular in signal processing and network systems to perform arithmetic over a window of inputs
- ▶ Useful in datapaths to implement register chains compactly
 - ▶ Xilinx 4-LUTs can be configured at 16-deep shift registers. The 16 SRAM cells can be cleverly wired to implement such memories

Shift Register Verilog (Interface)

```
module sreg #(
    parameter DEPTH=8,
    parameter DATAWIDTH=32
) (
    input wire clk,
    input wire rst,
    input wire [DATAWIDTH - 1:0] data_in,
    output wire [DATAWIDTH - 1:0] data_out,
    input wire shift_en
);

reg [DATAWIDTH - 1:0] mem[DEPTH - 1:0];
```

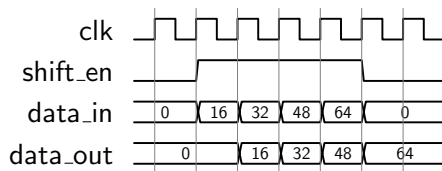
Shift Register Verilog (Loop)

```
always @(posedge clk) begin
  if(rst) begin
    for (i=0; i<=DEPTH-1; i=i+1) begin
      mem[i] <= 0;
    end
  end else begin
    if(shift_en) begin
      mem[0] <= data_in;
      for(i=0; i<DEPTH-1; i=i+1) begin
        mem[i+1] <= mem[i];
      end
    end
  end
end

assign data_out = mem[DEPTH-1];
```

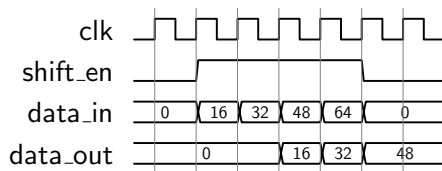
Shift Register Timing Diagram

DEPTH=1



Shift Register Timing Diagram

DEPTH=2

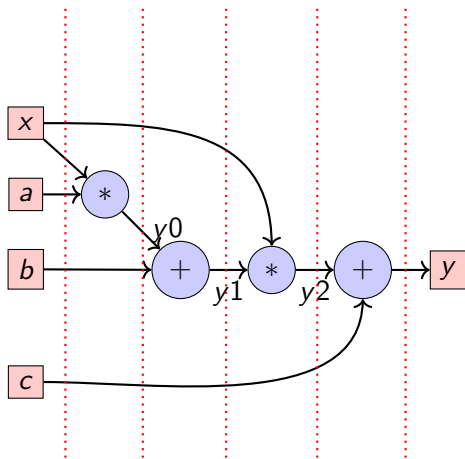


Shift Register RTL (Concatentation for 1b only!!)

```
reg [DEPTH - 1:0] mem;

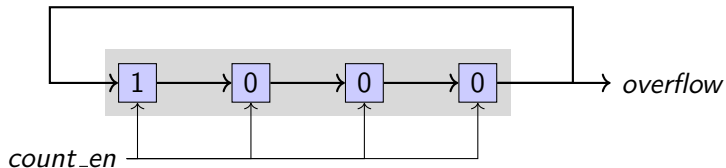
integer i;
always @(posedge clk) begin
  if(rst) begin
    for(i=0; i<DEPTH-1; i=i+1) begin
      mem[i] <= 0;
    end
  end else begin
    if(shift_en) begin
      mem[DEPTH-1:0] <= {mem[DEPTH - 2:0],data_in};
    end
  end
end
```


Shift Register Applications – Efficient Pipelining



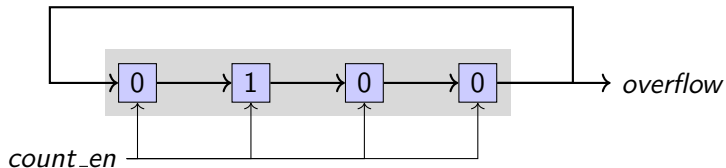
- ▶ Wire x is a shift register of DEPTH 2
- ▶ Wire c is a shift register of DEPTH 3

Shift Register Applications – One-hot counting



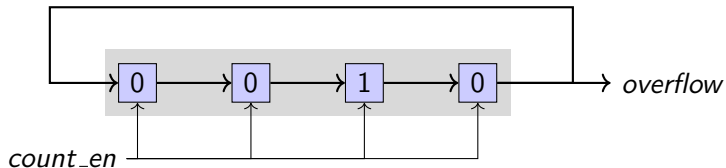
- ▶ Shift register can be configured as a counter
 - ▶ Binary counter needs $\log N$ bits, but a Shift register counter needs N bits!
- ▶ Initialized with a one-hot encoded assignment
- ▶ Final *overflow* bit is the timing signal used by developer
- ▶ Pattern rotates one cycle at a time → endless operation

Shift Register Applications – One-hot counting



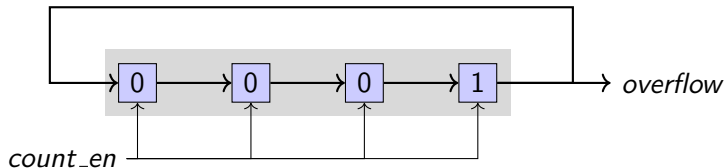
- ▶ Shift register can be configured as a counter
 - ▶ Binary counter needs $\log N$ bits, but a Shift register counter needs N bits!
- ▶ Initialized with a one-hot encoded assignment
- ▶ Final *overflow* bit is the timing signal used by developer
- ▶ Pattern rotates one cycle at a time → endless operation

Shift Register Applications – One-hot counting



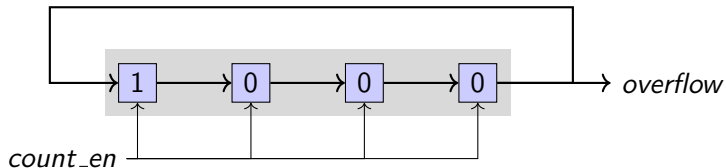
- ▶ Shift register can be configured as a counter
 - ▶ Binary counter needs $\log N$ bits, but a Shift register counter needs N bits!
- ▶ Initialized with a one-hot encoded assignment
- ▶ Final *overflow* bit is the timing signal used by developer
- ▶ Pattern rotates one cycle at a time → endless operation

Shift Register Applications – One-hot counting



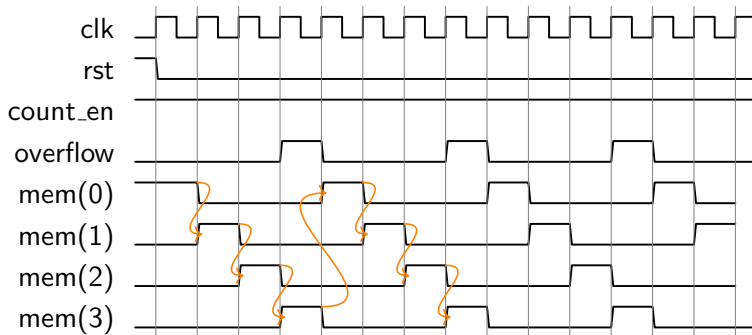
- ▶ Shift register can be configured as a counter
 - ▶ Binary counter needs $\log N$ bits, but a Shift register counter needs N bits!
- ▶ Initialized with a one-hot encoded assignment
- ▶ Final *overflow* bit is the timing signal used by developer
- ▶ Pattern rotates one cycle at a time → endless operation

Shift Register Applications – One-hot counting



- ▶ Shift register can be configured as a counter
 - ▶ Binary counter needs $\log N$ bits, but a Shift register counter needs N bits!
- ▶ Initialized with a one-hot encoded assignment
- ▶ Final *overflow* bit is the timing signal used by developer
- ▶ Pattern rotates one cycle at a time → endless operation

One-Hot Counter Timing Diagram

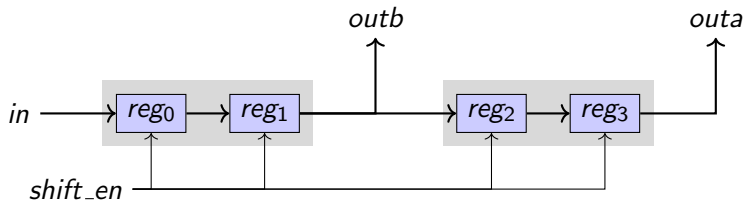


One-Hot Counter RTL

```
always @(posedge clk) begin
  if(rst) begin
    mem[DEPTH-1:1] <= 1'b0; // VERY IMPORTANT TO RESET CORRECTLY!
    mem[0] <= 1'b1;
  end else begin
    if(count_en) begin
      mem[DEPTH - 1:0] <= {mem[DEPTH - 2:0], mem[DEPTH - 1]};
    end
  end
end

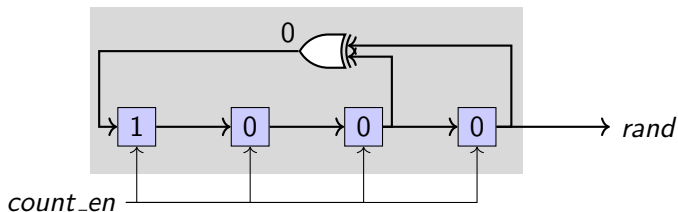
assign overflow = mem[DEPTH - 1];
```


Multi-tap Shift Registers



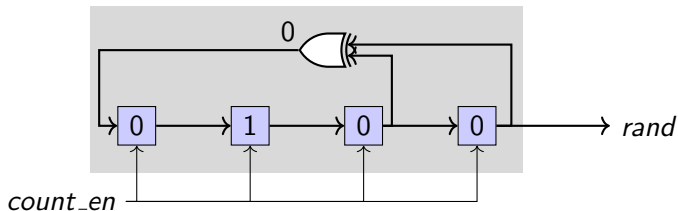
- ▶ Multi-tap shift registers useful when you want two or more copies of data from the past
- ▶ Effectively two back-to-back shift registers with the internal wire exposed for designer use
- ▶ Used in signal processing (FIR filters), random number generation, checksum computations

Linear Feedback Shift Registers



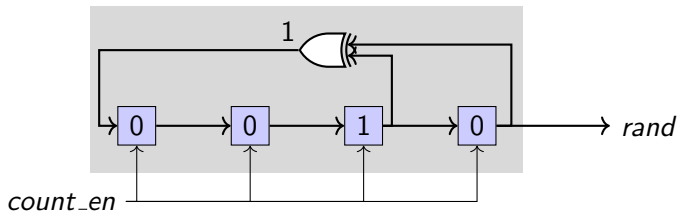
- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



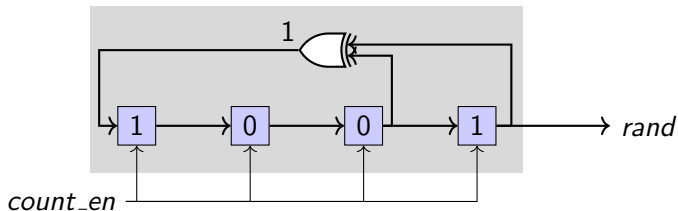
- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



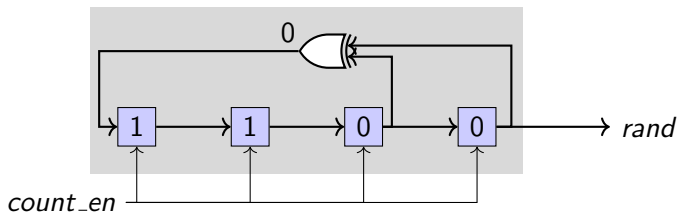
- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



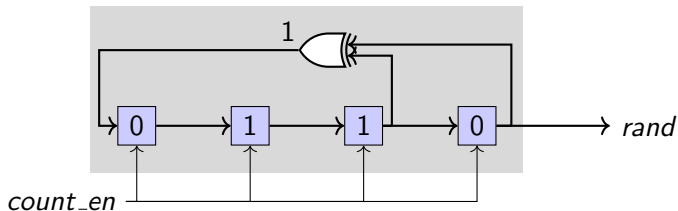
- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



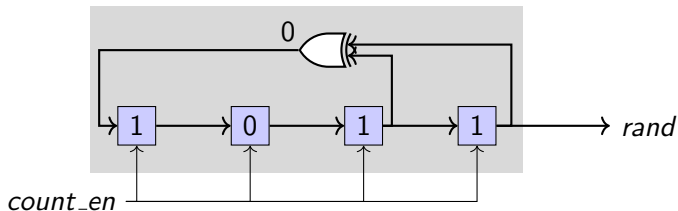
- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



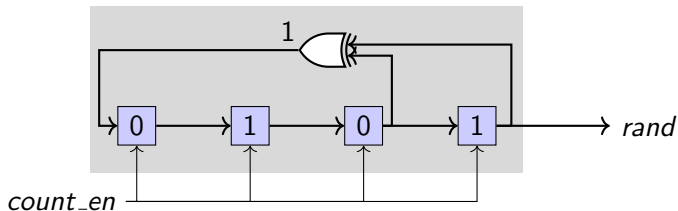
- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



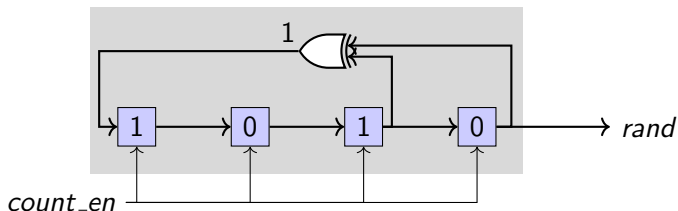
- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



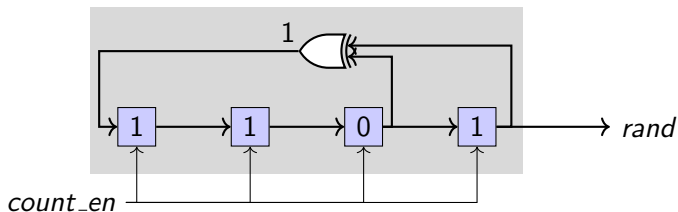
- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



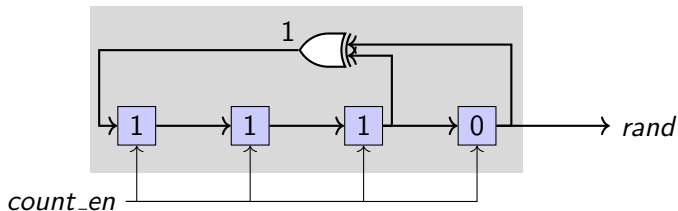
- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



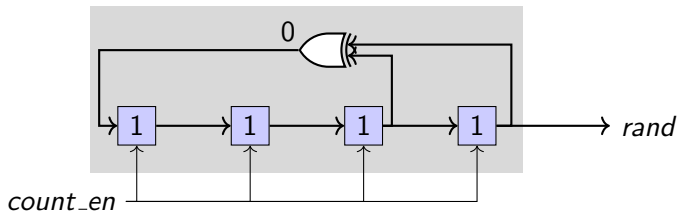
- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



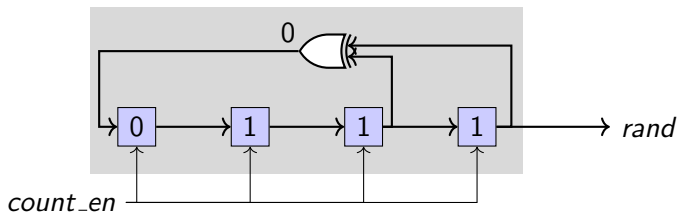
- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



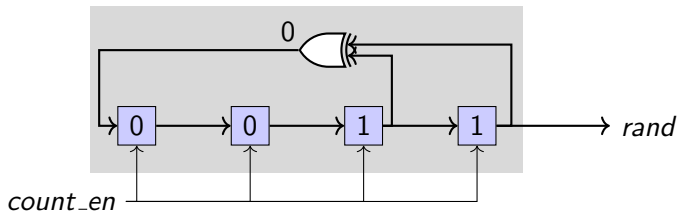
- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



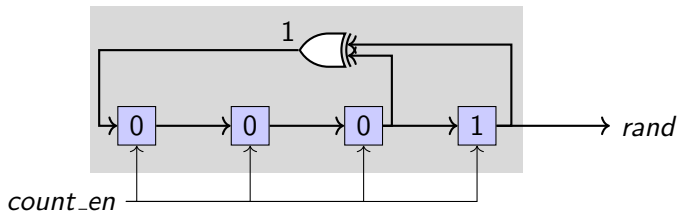
- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



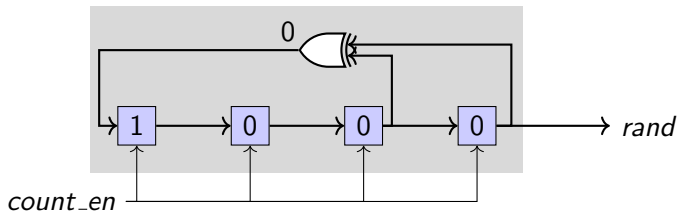
- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Linear Feedback Shift Registers



- ▶ Shift register can be configured as a random number generator
 - ▶ Generates pseudo-random numbers → not truly random, but cheap approximations
- ▶ LFSRs are generated based on specific polynomials that are known to generate randomness
 - ▶ Careful initialization after reset
 - ▶ XOR feedback required based on primitive polynomial
 - ▶ Incorrect initial value, or wrong XOR taps will not be a random pattern!

Pick LFSR feedback taps

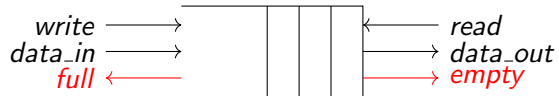
Cycle	Binary
0	1000
1	0100
2	0010
3	1001
4	1100
5	0110
6	1011
7	0101
8	1010
9	1101
10	1110
11	1111
12	0111
13	0011
14	0001
15	1000

- ▶ LFSR of k bits generates a sequence of 2^k bits
- ▶ Pattern is repetitive \rightarrow not truly random
- ▶ If pick k large enough, we can make the periodicity sufficiently large \rightarrow sufficiently random for some cases
- ▶ Unique set of XOR taps generate random patterns for a value of k .

FIFO Memories

- ▶ First-in First-out memories with fixed-size but variable latency depending on read/write throughput
- ▶ Enables parallel operation → used in software too
- ▶ **Key Idea:** Decouple producing and consumption of data inside a chip
- ▶ Ideal for connecting hardware components with loose timing behavior
- ▶ Decompose design into chunks that can be connected together in different ways

FIFO Symbol



Design of a FIFO

- ▶ FIFO is implemented as a memory + read/write address counters
 - ▶ On a write, increment write counter
 - ▶ On a read, increment read counter
 - ▶ Read can **never** race ahead of a write, or Write can **never** fall behind reads
- ▶ Dedicated port for writes, dedicated port for read
- ▶ Status indication flags for full and empty
 - ▶ Writer circuit needs to know if FIFO is full, if full no writes possible
 - ▶ Reader circuit needs to know if FIFO is empty, if empty no reads possible
- ▶ FIFO status flags computed by simple arithmetic on read and write counters

FIFO RTL

```
reg [ADDRWIDTH - 1:0] rdaddr;  
reg [ADDRWIDTH - 1:0] wraddr;  
reg [ADDRWIDTH - 1:0] occup;
```

```
mem #(  
    .ADDRWIDTH(ADDRWIDTH),  
    .DATAWIDTH(DATAWIDTH))  
mem_inst(  
    .clk(clk),  
    .rst(rst),  
    .rddata(data_out),  
    .wrdata(data_in),  
    .wraddr(wraddr),  
    .rdaddr(rdaddr),  
    .we(write));
```

```
if(rst) begin  
    rdaddr <= {ADDRWIDTH{1'b0}};  
    wraddr <= {ADDRWIDTH{1'b0}};  
    occup <= {ADDRWIDTH{1'b0}};  
end else begin  
    if(write) begin  
        wraddr <= wraddr + 1;  
    end  
    if(read) begin  
        rdaddr <= rdaddr + 1;  
    end  
    //  
    if(read && !write) begin  
        occup <= occup - 1;  
    end  
    else if(!read && write) begin  
        occup <= occup + 1;  
    end  
end  
end  
//  
assign full = occup == (DEPTH-1) ? 1'b1:1'b0;  
assign empty = occup == 0 ? 1'b1:1'b0;
```

FIFO RTL

```
reg [ADDRWIDTH - 1:0] rdaddr;  
reg [ADDRWIDTH - 1:0] wraddr;  
reg [ADDRWIDTH - 1:0] occup;
```

```
mem #(  
    .ADDRWIDTH(ADDRWIDTH),  
    .DATAWIDTH(DATAWIDTH))  
mem_inst(  
    .clk(clk),  
    .rst(rst),  
    .rddata(data_out),  
    .wrdata(data_in),  
    .wraddr(wraddr),  
    .rdaddr(rdaddr),  
    .we(write_en)
```

Watch out for unregistered
full and empty signals

```
if(rst) begin  
    rdaddr <= {ADDRWIDTH{1'b0}};  
    wraddr <= {ADDRWIDTH{1'b0}};  
    occup <= {ADDRWIDTH{1'b0}};
```

Missing
read && write
!read && !write

```
    rdaddr <= rdaddr + 1;  
end  
//  
if(read && !write) begin  
    occup <= occup - 1;  
end  
else if(!read && write) begin  
    occup <= occup + 1;  
end  
end  
end  
//  
assign full = occup == (DEPTH-1) ? 1'b1:1'b0;  
assign empty = occup == 0 ? 1'b1:1'b0;
```

FIFO Verilog (Effect of pipelining control)

- ▶ Unregistered full and empty signals.

```
assign full = occup == (DEPTH-1) ? 1'b1:1'b0;  
assign empty = occup == 0 ? 1'b1:1'b0;
```

- ▶ Registered full and empty signals require adjusting conditions for detection to be a cycle sooner.

Full detection with pipelining

```
if((occup==(DEPTH-1) || (occup==(DEPTH-2) & write & !read)) &  
    !(occup==(DEPTH-1) & read))  
begin  
    full <= 1'b1;  
end else begin  
    full <= 1'b0;  
end
```

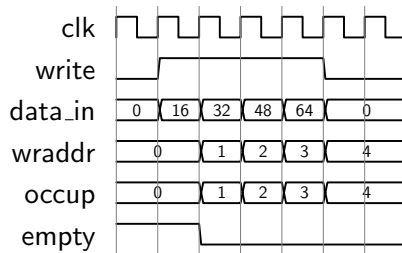
- ▶ Speculate that if we are one away from DEPTH-1 and a write is happening → we will be full in the next cycle.
- ▶ Speculate that if we are already full and a read is happening right now, → we will not be full in the next cycle.

Empty detection with pipelining

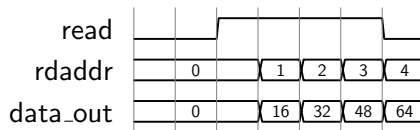
```
if((occup==0 || (occup==1 & read & !write)) &  
    !(occup==0 & write))  
begin  
    empty <= 1'b1;  
end else begin  
    empty <= 1'b0;  
end
```

- ▶ Speculate that if we are one away from 0 occupancy and a read is happening → we will be empty in the next cycle.
- ▶ Speculate that if we are already empty and a write is happening right now, → we will not be empty in the next cycle.

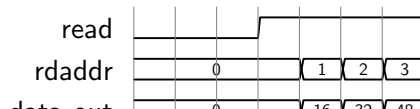
FIFO Timing Diagram (Self Reads)



```
assign read = !empty;
```

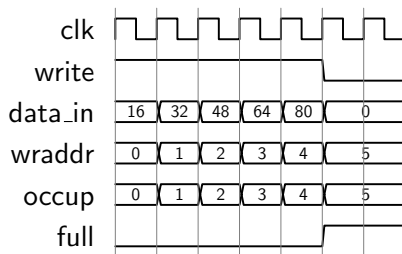


```
always @(posedge clk) read <= !empty;
```

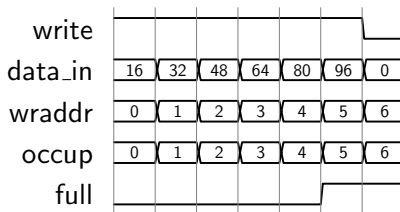


FIFO Timing Diagram (Self Writes)

```
assign write = !full;
```



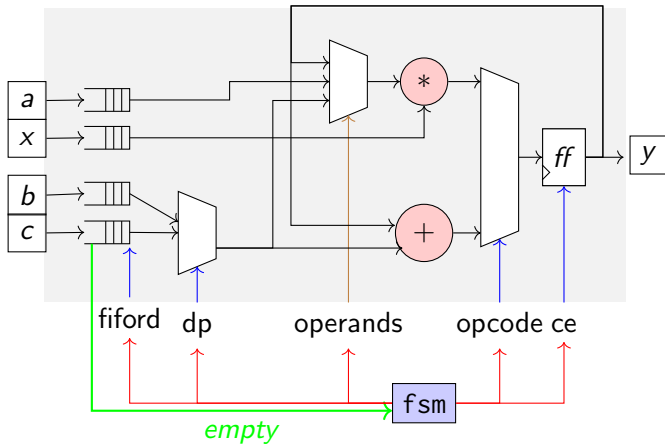
```
always@(posedge clk) write <= !full;
```



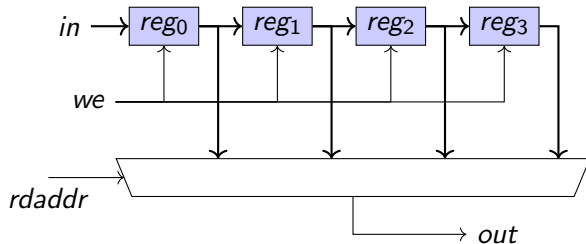
Notes on FIFO behavior

- ▶ Typically fifo full/empty signals need to be registered
- ▶ Read/write control signals are generated based on these status signals
- ▶ With pipelining, reads are often delayed by a minimum of three cycles after write
 - ▶ FWFT – First Word Fall Through mode to bypass storage for faster read (two cycles)
 - ▶ FWFT + combinational empty could reduce this to one cycle
- ▶ Early full/empty indications useful to logic that write/read from FIFO

Using FIFOs with poly (Resource-Shared)



FIFOs using Shift Registers



- ▶ On Xilinx devices, a LUT can be *programmed* as a shift register. For 4-LUTs, you get an incredible 16-deep shift register in just one LUT
- ▶ LUT inputs can be repurposed to implement programmable **tap** into the shift register if you want depths < 16 .
- ▶ Can implement small FIFO compactly by locking down write location to address 0 and adjusting tap
 - ▶ `wraddr=0` always
 - ▶ LUT inputs = `rdaddr`

Loop Unrolling $y = a \times x + b$

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

Cycle	Operators	
	<i>add₀</i>	<i>mult₀</i>
0	-	$reg_0 = a \cdot x$
1	$y = reg_0 + b$	-

- ▶ Scheduling often leaves resources idle!
What is data available?
- ▶ **Loop Unrolling** is a technique that flattens multiple loop iterations into a large loop body

Loop Unrolling $y = a \times x + b$

```
for(int i=0;i<N;i+=2) {  
  y[i] = a[i]*x[i] + b[i];  
  y[i+1] = a[i+1]*x[i+1] + b[i+1];  
}
```

Cycle	Operators	
	<i>add₀</i>	<i>mult₀</i>
0	-	$reg_0 = a_0 \cdot x_0$
1	$y_0 = reg_0 + b_0$	$reg_0 = a_1 \cdot x_1$
2	$y_1 = reg_0 + b_1$	-

- ▶ Scheduling often leaves resources idle!
What is data available?
- ▶ **Loop Unrolling** is a technique that flattens multiple loop iterations into a large loop body

Loop Unrolling $y = a \times x + b$

```
for(int i=0;i<N;i+=2) {  
  y[i] = a[i]*x[i] + b[i];  
  y[i+1] = a[i+1]*x[i+1] + b[i+1];  
}
```

Cycle	Operators	
	<i>add</i> ₀	<i>mult</i> ₀
0	-	$reg_0 = a_0 \cdot x_0$
1	$y_0 = reg_0 + b_0$	$reg_0 = a_1 \cdot x_1$
2	$y_1 = reg_0 + b_1$	-

- ▶ Scheduling often leaves resources idle!
What is data available?
- ▶ **Loop Unrolling** is a technique that flattens multiple loop iterations into a large loop body

Loop Unrolling $y = a \times x + b$

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

Cycle	Operators	
	<i>add</i> ₀	<i>mult</i> ₀
0	-	$reg_0 = a_0 \cdot x_0$
1	$y_0 = reg_0 + b_0$	$reg_0 = a_1 \cdot x_1$
2	$y_1 = reg_0 + b_1$	$reg_0 = a_2 \cdot x_2$
3	$y_2 = reg_0 + b_2$	$reg_0 = a_3 \cdot x_3$
4	$y_3 = reg_0 + b_3$	-

- ▶ More freedom to the scheduler → reduces waste ↓
- ▶ Increases intermediate *reg* cost ↑
- ▶ Increases complexity of scheduler state machine ↑

Application to poly

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

Cycle	Operators	
	<i>add₀</i>	<i>mult₀</i>
0	-	$reg_0 = a \cdot x$
1	$reg_0 = reg_0 + b$	-
2	-	$reg_0 = reg_0 \cdot x$
3	$y = reg_0 + c$	-

- ▶ If solving $y = a \cdot x^2 + b \cdot x + c$, we can factor as $(a \cdot x + b) \cdot x + c$
- ▶ Recall, unrolling requires more intermediate state
- ▶ Simple solution needs multiple intermediate registers for each intermediate hop.
- ▶ Can scavenge/optimize to reduce number of registers

Application to poly

```
for(int i=0;i<N;i+=2) {  
  y[i] = (a[i]*x[i] + b[i])*x[i]+c[i];  
  y[i+1] = (a[i+1]*x[i+1] + b[i+1])*x[i+1]+c[i+1];  
}
```

Cycle	Operators	
	<i>add</i> ₀	<i>mult</i> ₀
0	-	$reg_0 = a_0 \cdot x_0$
1	$reg_1 = reg_0 + b_0$	$reg_0 = a_1 \cdot x_1$
2	$reg_1 = reg_0 + b_1$	$reg_2 = reg_1 \cdot x_0$
3	$y_0 = reg_2 + c_0$	$reg_2 = reg_1 \cdot x_1$
4	$y_1 = reg_2 + c_1$	-

- ▶ If solving $y = a \cdot x^2 + b \cdot x + c$, we can factor as $(a \cdot x + b) \cdot x + c$
- ▶ Recall, unrolling requires more intermediate state
- ▶ Simple solution needs multiple intermediate registers for each intermediate hop.
- ▶ Can scavenge/optimize to reduce number of registers

Application to poly

```
for(int i=0;i<N;i+=2) {  
  y[i] = (a[i]*x[i] + b[i])*x[i]+c[i];  
  y[i+1] = (a[i+1]*x[i+1] + b[i+1])*x[i+1]+c[i+1];  
}
```

Cycle	Operators	
	<i>add</i> ₀	<i>mult</i> ₀
0	-	$reg_0 = a_0 \cdot x_0$
1	$reg_1 = reg_0 + b_0$	$reg_0 = a_1 \cdot x_1$
2	$reg_1 = reg_0 + b_1$	$reg_2 = reg_1 \cdot x_0$
3	$y_0 = reg_2 + c_0$	$reg_2 = reg_1 \cdot x_1$
4	$y_1 = reg_2 + c_1$	-

- ▶ If solving $y = a \cdot x^2 + b \cdot x + c$, we can factor as $(a \cdot x + b) \cdot x + c$
- ▶ Recall, unrolling requires more intermediate state
- ▶ Simple solution needs multiple intermediate registers for each intermediate hop.
- ▶ Can scavenge/optimize to reduce number of registers

Application to poly

```
for(int i=0;i<N;i+=2) {  
  y[i] = (a[i]*x[i] + b[i])*x[i]+c[i];  
  y[i+1] = (a[i+1]*x[i+1] + b[i+1])*x[i+1]+c[i+1];  
}
```

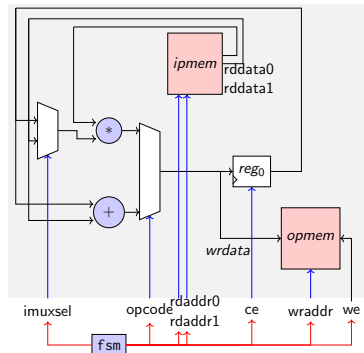
Cycle	Operators	
	<i>add</i> ₀	<i>mult</i> ₀
0	-	$reg_0 = a_0 \cdot x_0$
1	$reg_1 = reg_0 + b_0$	$reg_0 = a_1 \cdot x_1$
2	$reg_1 = reg_0 + b_1$	$reg_0 = reg_1 \cdot x_0$
3	$y_0 = reg_0 + c_0$	$reg_0 = reg_1 \cdot x_1$
4	$y_1 = reg_0 + c_1$	-

- ▶ If solving $y = a \cdot x^2 + b \cdot x + c$, we can factor as $(a \cdot x + b) \cdot x + c$
- ▶ Recall, unrolling requires more intermediate state
- ▶ Simple solution needs multiple intermediate registers for each intermediate hop.
- ▶ Can scavenge/optimize to reduce number of registers

Unroll + S/W Pipe Datapath Diagram for poly

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

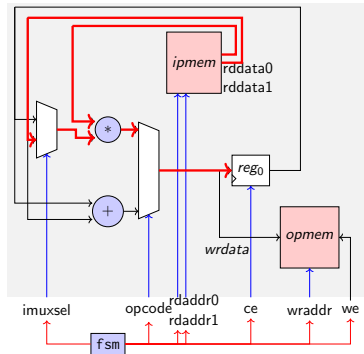
Cycle	Operators	
	add_0	$mult_0$
0	-	$reg_0 = a \cdot x$
1	$reg_0 = reg_0 + b$	-
2	-	$reg_0 = reg_0 \cdot x$
3	$y = reg_0 + c$	-



Unroll + S/W Pipe Datapath Diagram for poly

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

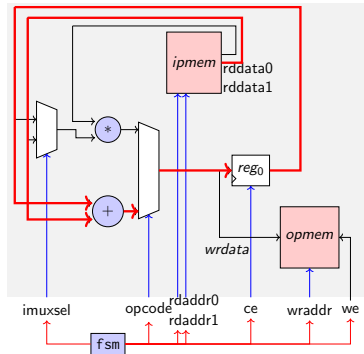
Cycle	Operators	
	add_0	$mult_0$
→ 0	-	$reg_0 = a \cdot x$
1	$reg_0 = reg_0 + b$	-
2	-	$reg_0 = reg_0 \cdot x$
3	$y = reg_0 + c$	-



Unroll + S/W Pipe Datapath Diagram for poly

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

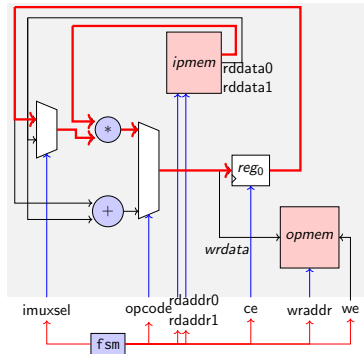
Cycle	Operators	
	add_0	$mult_0$
0	-	$reg_0 = a \cdot x$
→ 1	$reg_0 = reg_0 + b$	-
2	-	$reg_0 = reg_0 \cdot x$
3	$y = reg_0 + c$	-



Unroll + S/W Pipe Datapath Diagram for poly

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

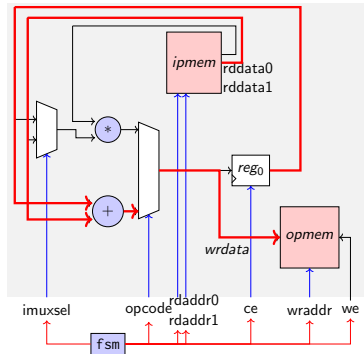
Cycle	Operators	
	add_0	$mult_0$
0	-	$reg_0 = a \cdot x$
1	$reg_0 = reg_0 + b$	-
→ 2	-	$reg_0 = reg_0 \cdot x$
3	$y = reg_0 + c$	-



Unroll + S/W Pipe Datapath Diagram for poly

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

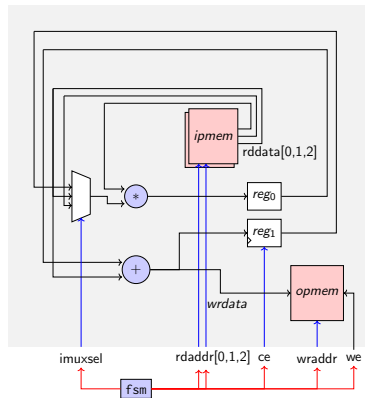
Cycle	Operators	
	add_0	$mult_0$
0	-	$reg_0 = a \cdot x$
1	$reg_0 = reg_0 + b$	-
2	-	$reg_0 = reg_0 \cdot x$
→ 3	$y = reg_0 + c$	-



Unroll + S/W Pipe Datapath Diagram for poly

```
for(int i=0;i<N;i+=2) {
  y[i] = (a[i]*x[i] + b[i])*x[i]+c[i];
  y[i+1] = (a[i+1]*x[i+1] + b[i+1])*x[i+1]+c[i+1];
}
```

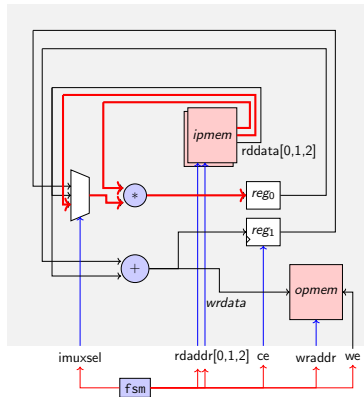
Cycle	Operators	
	<i>add₀</i>	<i>mult₀</i>
0	-	$reg_0 = a_0 \cdot x_0$
1	$reg_1 = reg_0 + b_0$	$reg_0 = a_1 \cdot x_1$
2	$reg_1 = reg_0 + b_1$	$reg_0 = reg_1 \cdot x_0$
3	$y_0 = reg_0 + c_0$	$reg_0 = reg_1 \cdot x_1$
4	$y_1 = reg_0 + c_1$	-



Unroll + S/W Pipe Datapath Diagram for poly

```
for(int i=0;i<N;i+=2) {
  y[i] = (a[i]*x[i] + b[i])*x[i]+c[i];
  y[i+1] = (a[i+1]*x[i+1] + b[i+1])*x[i+1]+c[i+1];
}
```

Cycle	Operators	
	<i>add₀</i>	<i>mult₀</i>
→ 0	-	$reg_0 = a_0 \cdot x_0$
1	$reg_1 = reg_0 + b_0$	$reg_0 = a_1 \cdot x_1$
2	$reg_1 = reg_0 + b_1$	$reg_0 = reg_1 \cdot x_0$
3	$y_0 = reg_0 + c_0$	$reg_0 = reg_1 \cdot x_1$
4	$y_1 = reg_0 + c_1$	-

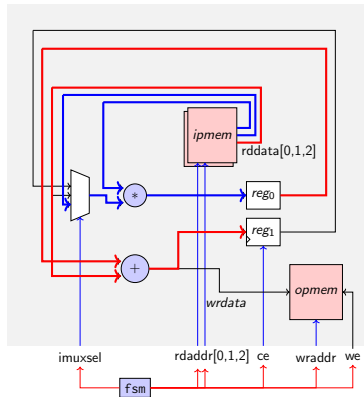


Unroll + S/W Pipe Datapath Diagram for poly

```

for(int i=0;i<N;i+=2) {
  y[i] = (a[i]*x[i] + b[i])*x[i]+c[i];
  y[i+1] = (a[i+1]*x[i+1] + b[i+1])*x[i+1]+c[i+1];
}
    
```

Cycle	Operators	
	<i>add₀</i>	<i>mult₀</i>
0	-	$reg_0 = a_0 \cdot x_0$
→ 1	$reg_1 = reg_0 + b_0$	$reg_0 = a_1 \cdot x_1$
2	$reg_1 = reg_0 + b_1$	$reg_0 = reg_1 \cdot x_0$
3	$y_0 = reg_0 + c_0$	$reg_0 = reg_1 \cdot x_1$
4	$y_1 = reg_0 + c_1$	-

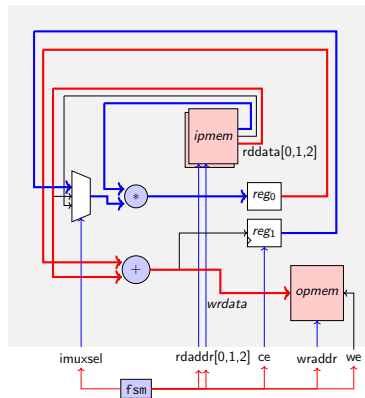


Unroll + S/W Pipe Datapath Diagram for poly

```

for(int i=0;i<N;i+=2) {
  y[i] = (a[i]*x[i] + b[i])*x[i]+c[i];
  y[i+1] = (a[i+1]*x[i+1] + b[i+1])*x[i+1]+c[i+1];
}
    
```

Cycle	Operators	
	<i>add₀</i>	<i>mult₀</i>
0	-	$reg_0 = a_0 \cdot x_0$
1	$reg_1 = reg_0 + b_0$	$reg_0 = a_1 \cdot x_1$
2	$reg_1 = reg_0 + b_1$	$reg_0 = reg_1 \cdot x_0$
→ 3	$y_0 = reg_0 + c_0$	$reg_0 = reg_1 \cdot x_1$
4	$y_1 = reg_0 + c_1$	-

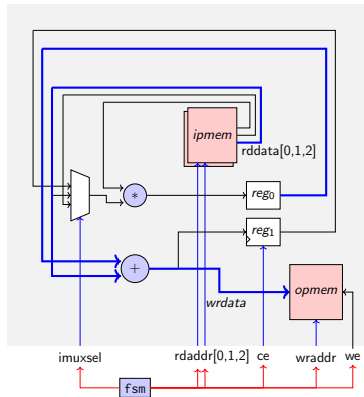


Unroll + S/W Pipe Datapath Diagram for poly

```

for(int i=0;i<N;i+=2) {
  y[i] = (a[i]*x[i] + b[i])*x[i]+c[i];
  y[i+1] = (a[i+1]*x[i+1] + b[i+1])*x[i+1]+c[i+1];
}
    
```

Cycle	Operators	
	<i>add</i> ₀	<i>mult</i> ₀
0	-	$reg_0 = a_0 \cdot x_0$
1	$reg_1 = reg_0 + b_0$	$reg_0 = a_1 \cdot x_1$
2	$reg_1 = reg_0 + b_1$	$reg_0 = reg_1 \cdot x_0$
3	$y_0 = reg_0 + c_0$	$reg_0 = reg_1 \cdot x_1$
→ 4	$y_1 = reg_0 + c_1$	-



Wrapup

- ▶ Memories can be combined in interesting ways to create multi-ported structures, FIFOs, and shift registers
- ▶ Focus on cheap, low-cost specialization to build each structure
- ▶ FIFOs are a common design feature in most chips with bursty, dynamic data arrival and compute patterns
- ▶ Unrolling of loops can help boost scheduling performance