Memories: a practical primer

- The good news: huge selection of technologies
  - Small & faster vs. large & slower
  - Every year capacities go up and prices go down
  - New kid on the block: high density, fast flash memories
    - Non-volatile, read/write, no moving parts! (robust, efficient)
- The bad news: perennial system bottleneck
  - Latencies (access time) haven't kept pace with cycle times
  - Separate technology from logic, so must communicate between silicon, so physical limitations (# of pins, R's and C's and L's) limit bandwidths
    - New hopes: capacitive interconnect, 3D IC's
  - Likely the limiting factor in cost & performance of many digital systems: designers spend a lot of time figuring out how to keep memories running at peak bandwidth
  - "It's the memory, stupid"

Memories in Verilog

- `reg bit;  // a single register`
- `reg [31:0] word;  // a 32-bit register`
- `reg [31:0] array[15:0]  // 16 32-bit regs`

- ```
  wire [31:0] read_data,write_data;
  wire [3:0] index;

  // combinational (asynch) read
  assign read_data = array[index];

  // clocked (synchronous) write
  always @ (posedge clock)
      array[index] <= write_data;
  ```
  ```

Multi-port Memories (aka regfiles)

```
reg [31:0] regfile[31:0]; // 32 32-bit words

// Beta register file: 2 read ports, 1 write
wire [4:0] ra1,ra2,wa;
wire [31:0] rd1,rd2,wd;

assign ra1 = inst[20:16];
assign ra2 = ra2sel ? inst[25:21] : inst[15:11];
assign wa = wasel ? 5'd30 : inst[25:21];

// read ports
assign rd1 = (ra1 == 31) ? 0 : regfile[ra1];
assign rd2 = (ra2 == 31) ? 0 : regfile[ra2];
// write port
always @ (posedge clk)
  if (werf) regfile[wa] <= wd;

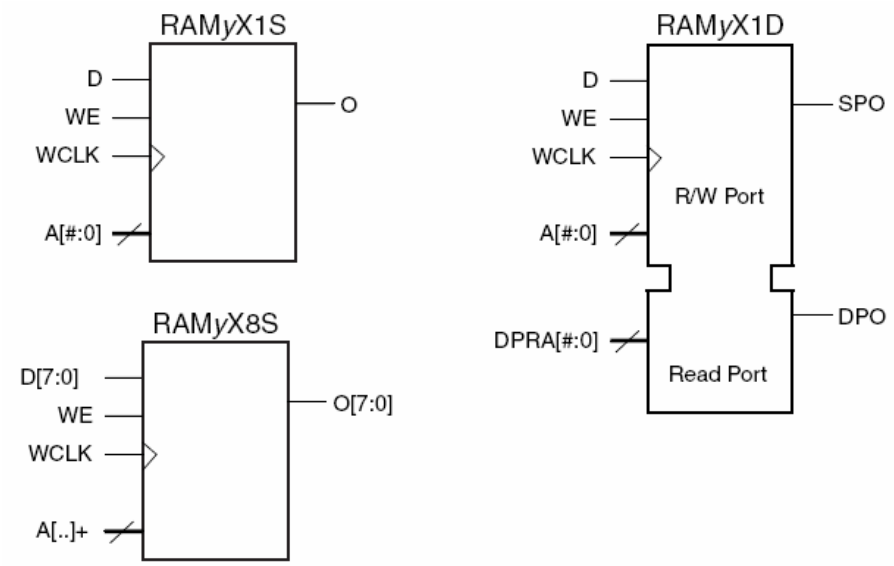
assign z = ~| rd1;   // used in BEQ/BNE instructions
```

FPGA memory implementation

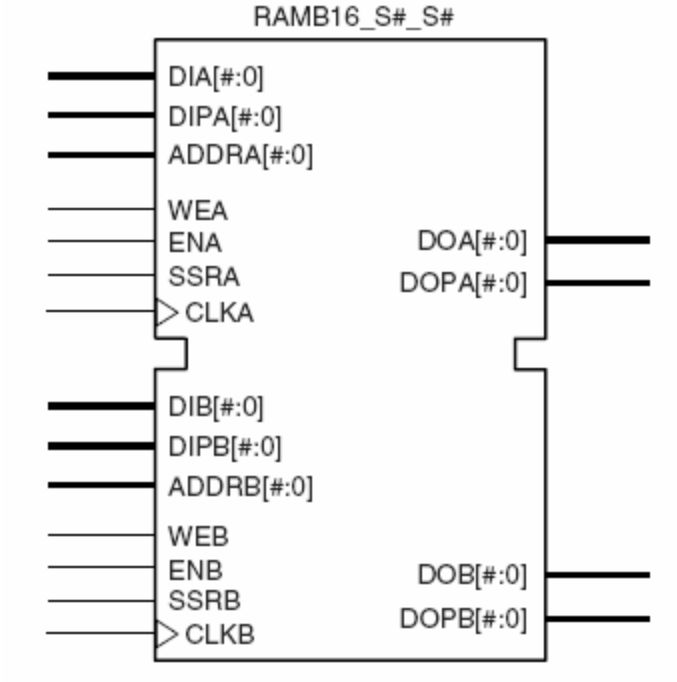
- Regular registers in logic blocks
  - Piggy use of resources, but convenient & fast if small
- [Xilinx Vertex II] use the LUTs:
  - Single port: 16x(1,2,4,8), 32x(1,2,4,8), 64x(1,2), 128x1
  - Dual port (1 R/W, 1R): 16x1, 32x1, 64x1
  - Can fake extra read ports by cloning memory: all clones are written with the same addr/data, but each clone can have a different read address
- [Xilinx Vertex II] use block ram:
  - 18K bits: 16Kx1, 8Kx2, 4Kx4
    with parity: 2Kx(8+1), 1Kx(16+2), 512x(32+4)
  - Single or dual port
  - Pipelined (clocked) operations
  - Labkit XCV2V6000: 144 BRAMs, 2952K bits total

Virtex memory configurations

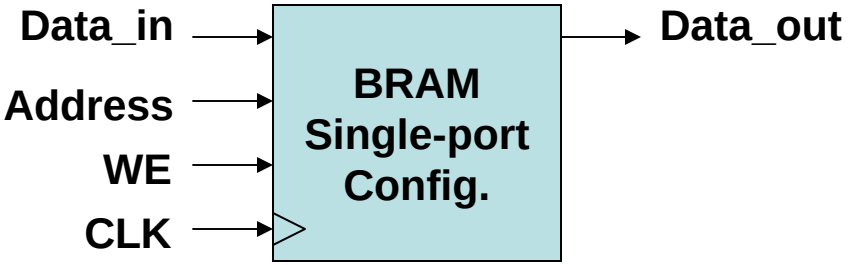
Using LUT resources in configurable logic blocks:

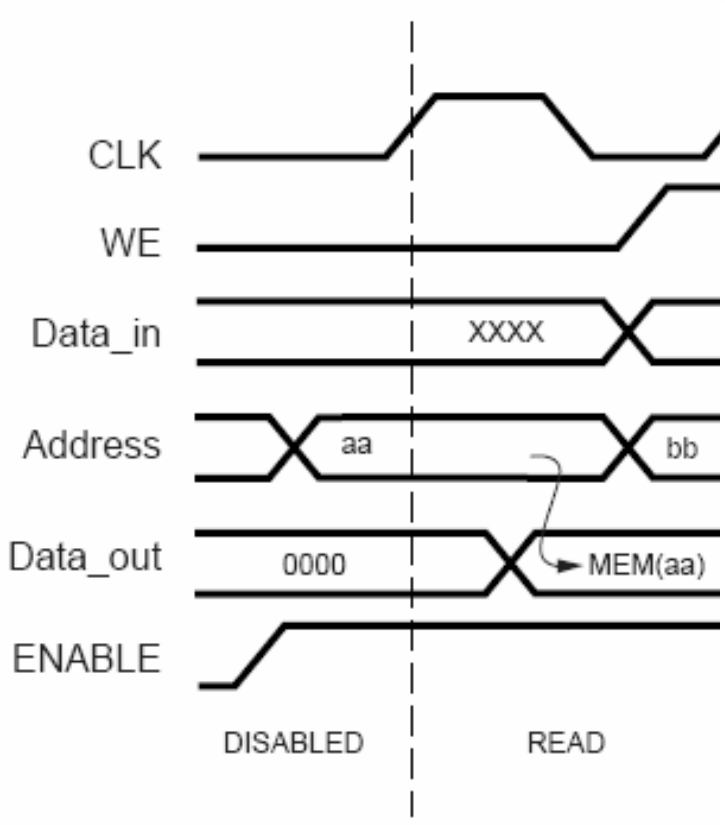


Using BRAMs:



Xilinx Synchronous Block Memory

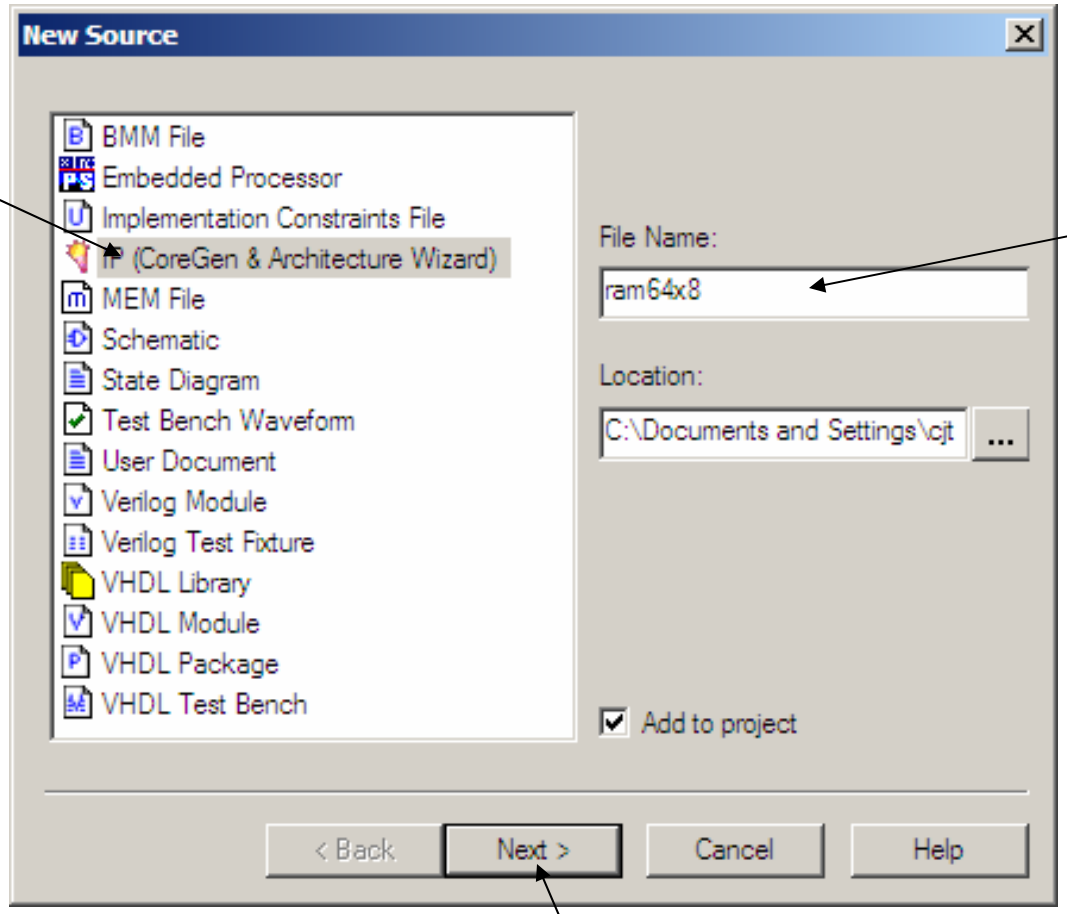




Source: Xilinx App Note 463

Using BRAMs (eg, a 64Kx8 ram)

- From menus: Project → New Source…



Select "IP"

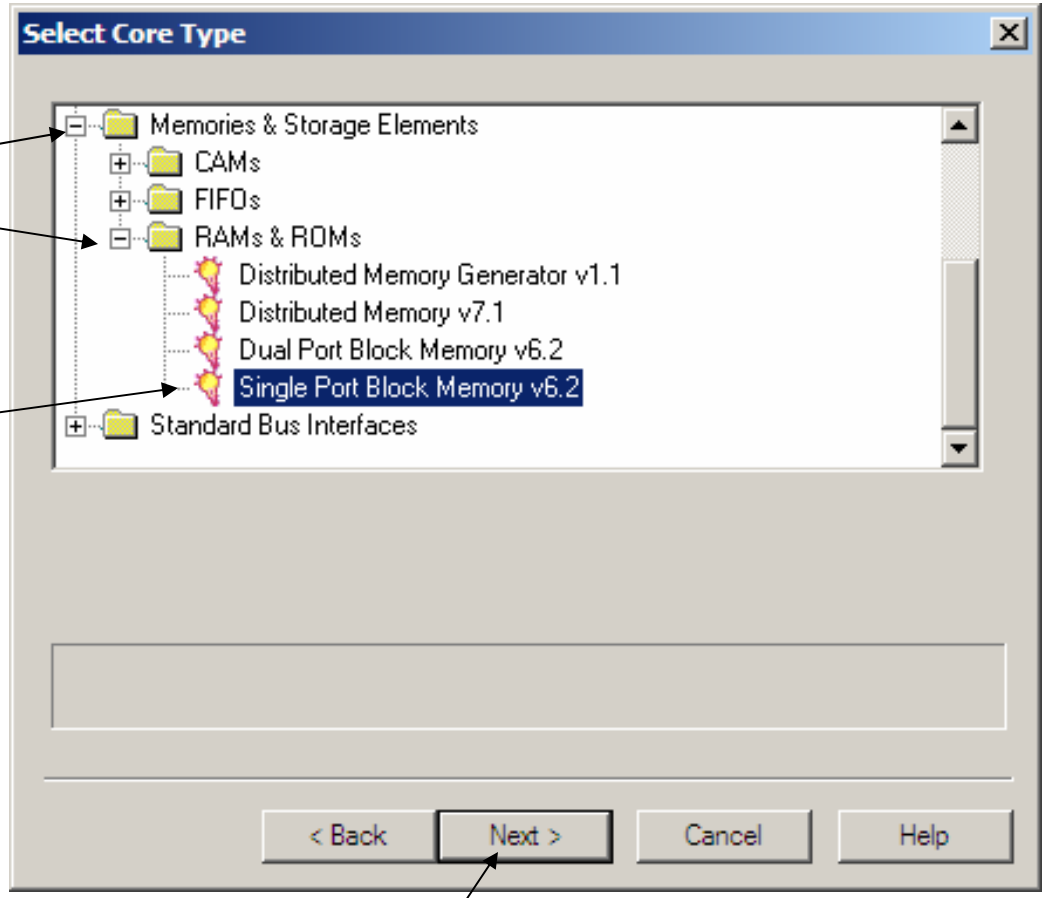
Fill in name

Click "Next" when done…

BRAM Example

Click open folders

Select "Single Port Block Memory"



Click "Next" and then "Finish" on next window

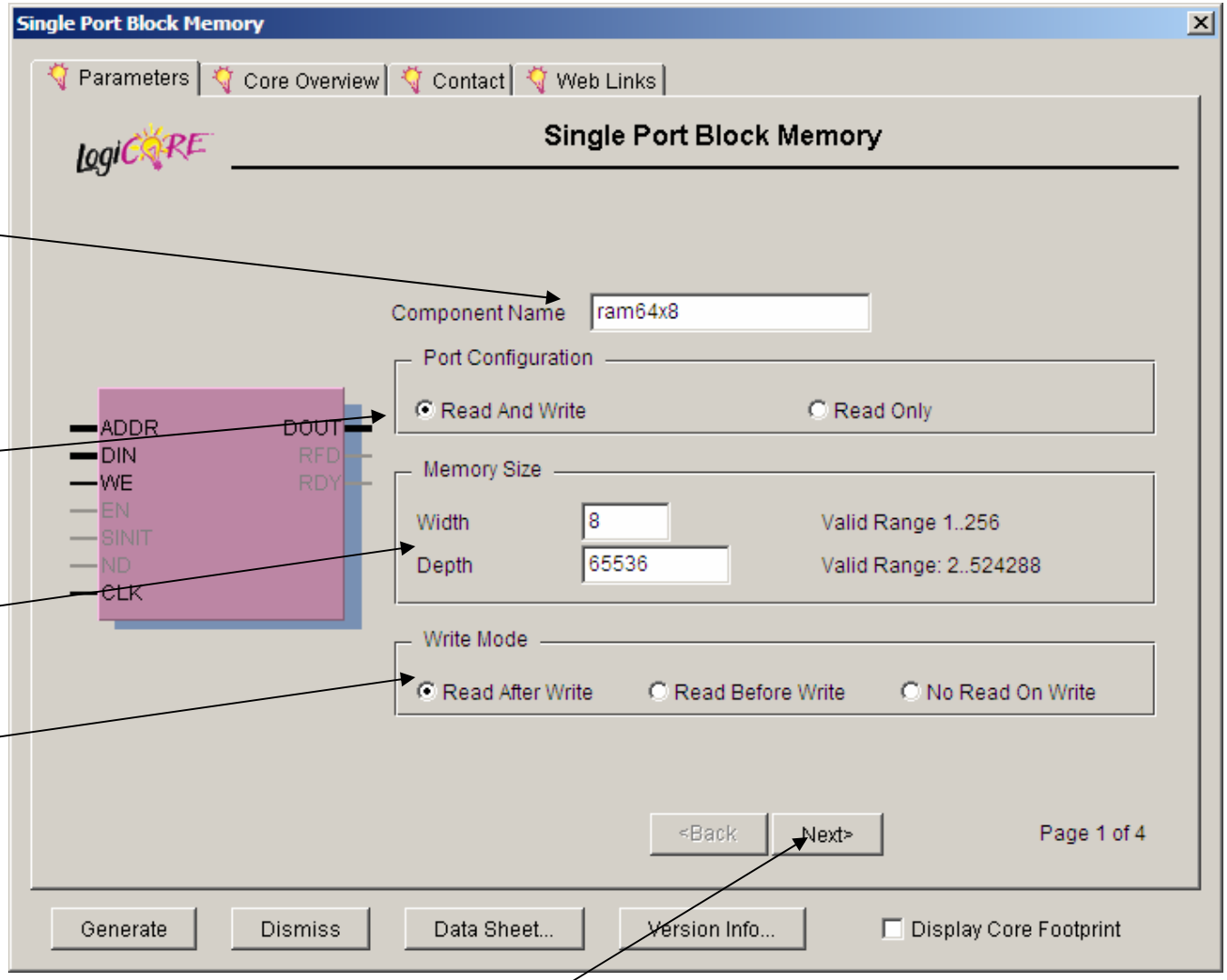
BRAM Example

Fill in name (again?!)

Select RAM vs ROM

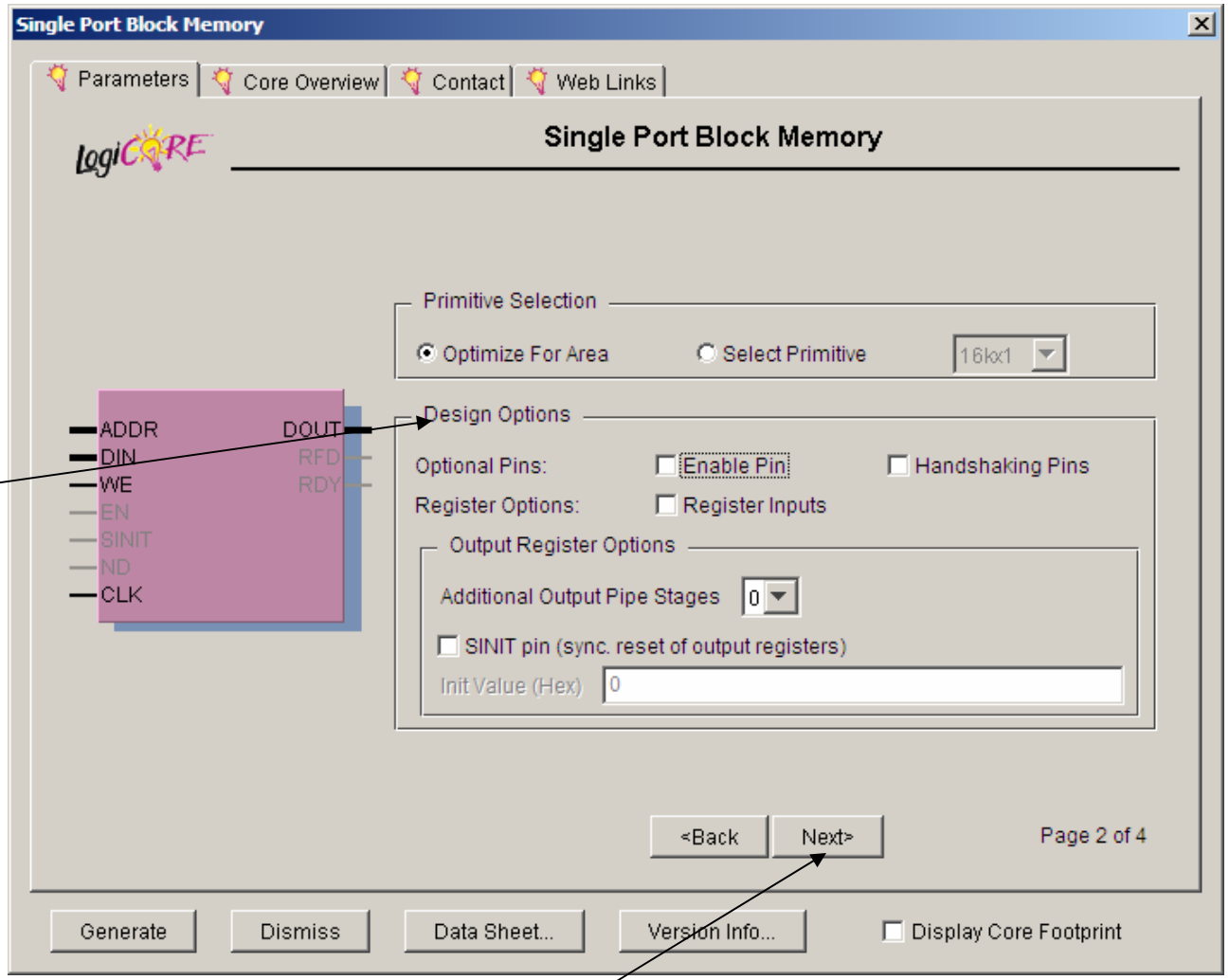
Fill in width & depth

Usually "Read After Write" is what you want



Click "Next" …

BRAM Example

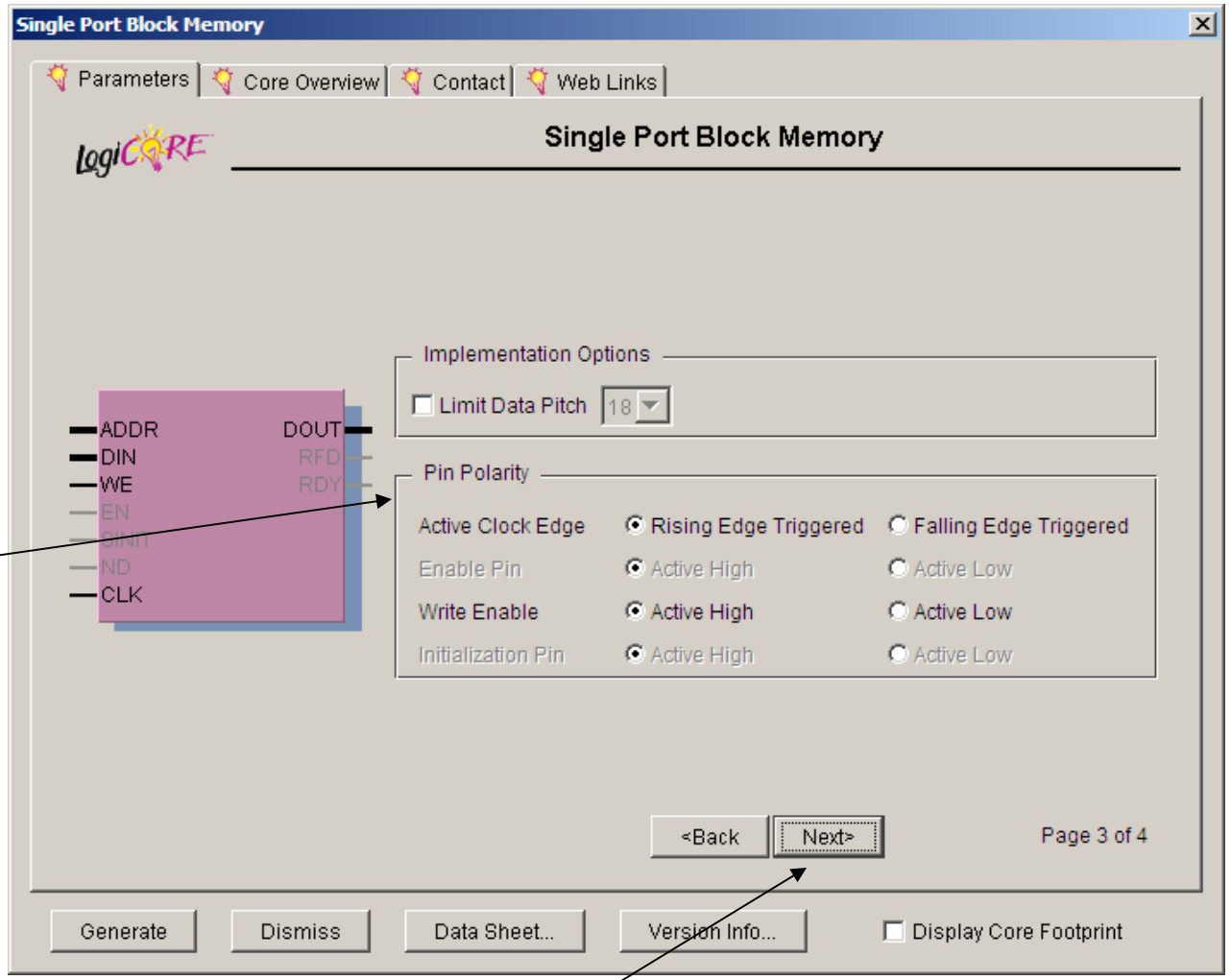


Can add extra control pins, but usually not

Click "Next" …

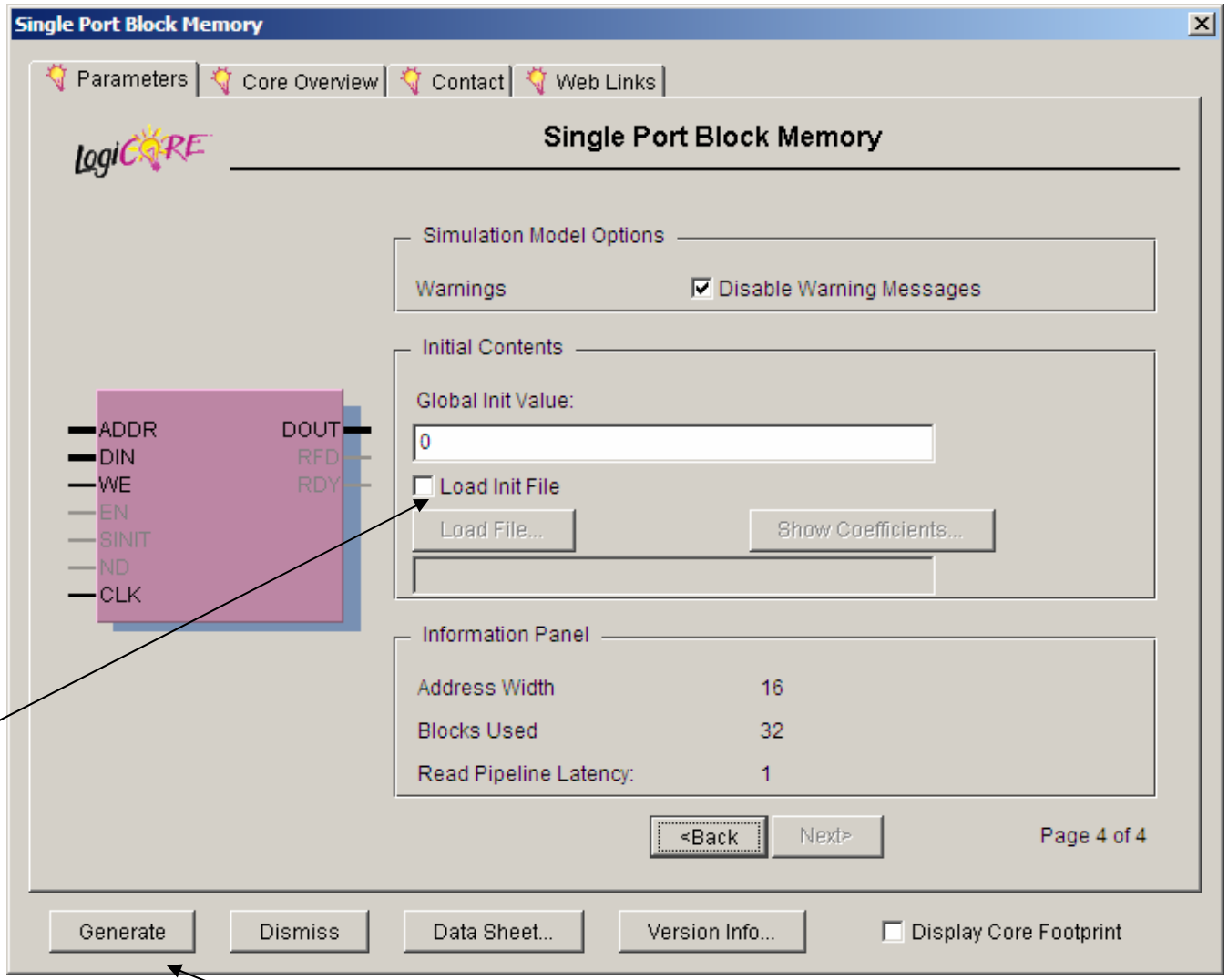
BRAM Example

Select polarity of control pins; active high default is usually just fine



Click "Next" …

BRAM Example



Click to name a .coe file that specifies initial contents (eg, for a ROM)

Click "Generate" to complete

.coe file format

```
memory_initialization_radix=2;
memory_initialization_vector=

00000000,
00111110,
01100011,
00000011,
00000011,
00011110,
00000011,
00000011,
01100011,
00111110,
00000000,
00000000,
```

Memory contents with location 0 first, then location 1, etc. You can specify input radix, in this example we're using binary. MSB is on the left, LSB on the right. Unspecified locations (if memory has more locations than given in .coe file) are set to 0.

Using result in your Verilog

- Look at generated Verilog for module def'n:

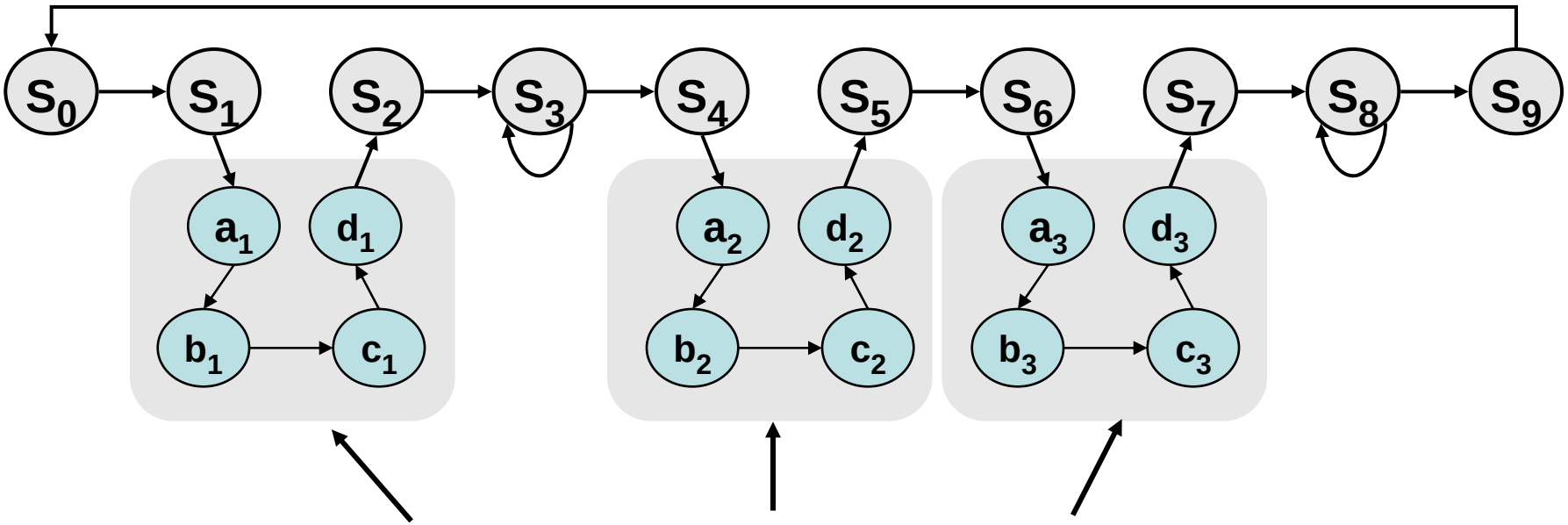
```
module ram64x8 (addr,clk,din,dout,we);
    input [15 : 0] addr;
    input clk;
    input [7 : 0] din;
    output [7 : 0] dout;
    input we;
    …
endmodule
```

- Use to instantiate instances in your code:

```
ram64x8 foo(addr,clk,din,dout,we);
```

Toward FSM Modularity

- Consider the following abstract FSM:

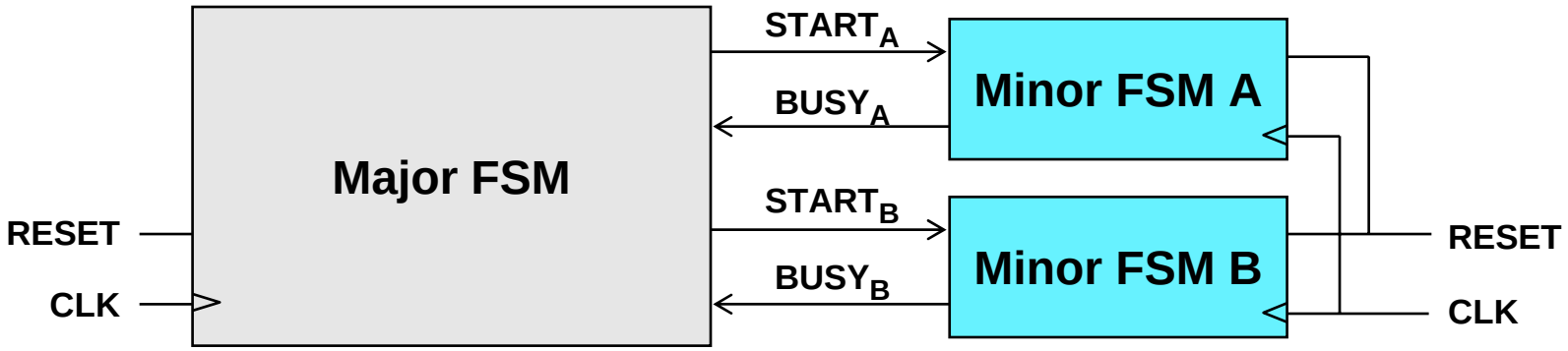


- Suppose that each set of states $a_x...d_x$ is a "sub-FSM" that produces exactly the same outputs.
- Can we simplify the FSM by removing equivalent states?

  No! The outputs may be the same, but the next-state transitions are not.
- This situation closely resembles a procedure call or function call in software...how can we apply this concept to FSMs?

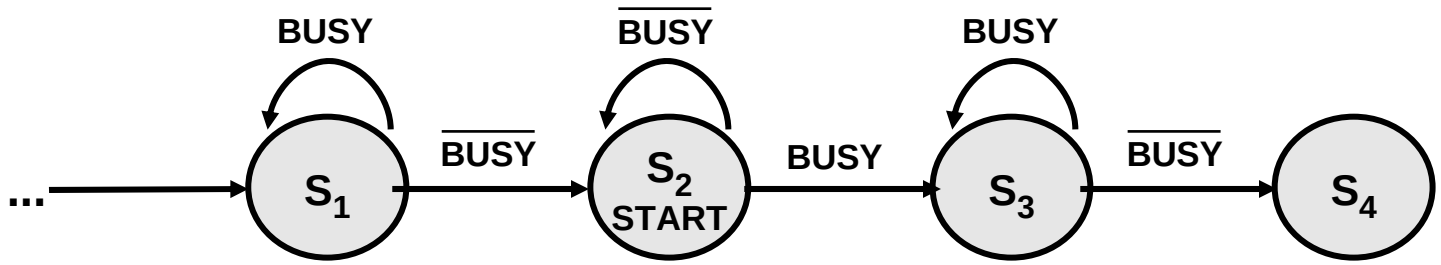
Acknowledgements: Rex Min

The Major/Minor FSM Abstraction



- Subtasks are encapsulated in minor FSMs with common reset and clock
- Simple communication abstraction:
  - START: tells the minor FSM to begin operation (the call)
  - BUSY: tells the major FSM whether the minor is done (the return)
- The major/minor abstraction is great for...
  - Modular designs (*always* a good thing)
  - Tasks that occur often but in different contexts
  - Tasks that require a variable/unknown period of time
  - Event-driven systems

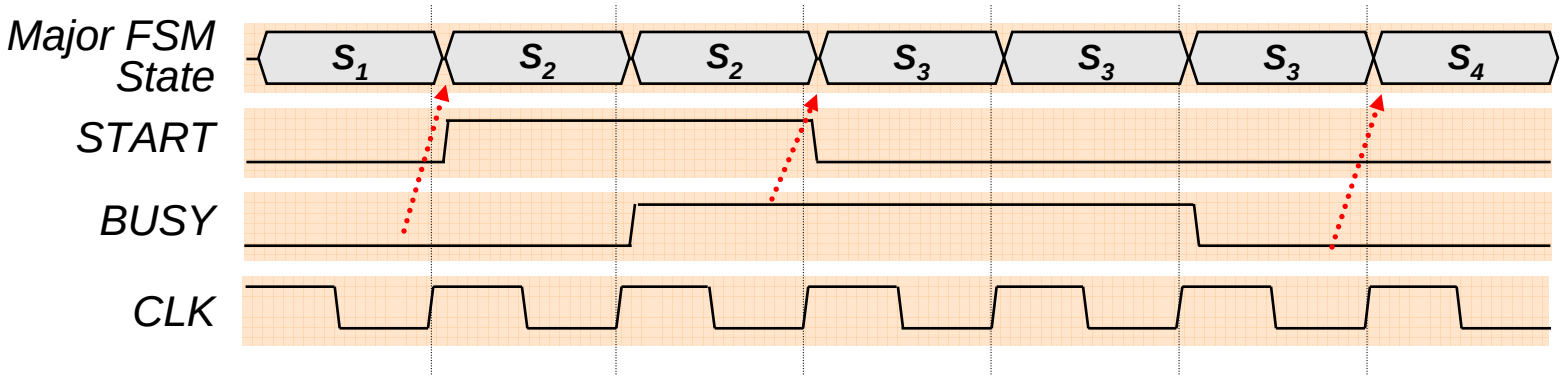
Inside the Major FSM



1. Wait until the minor FSM is ready

2. Trigger the minor FSM (and make sure it's started)

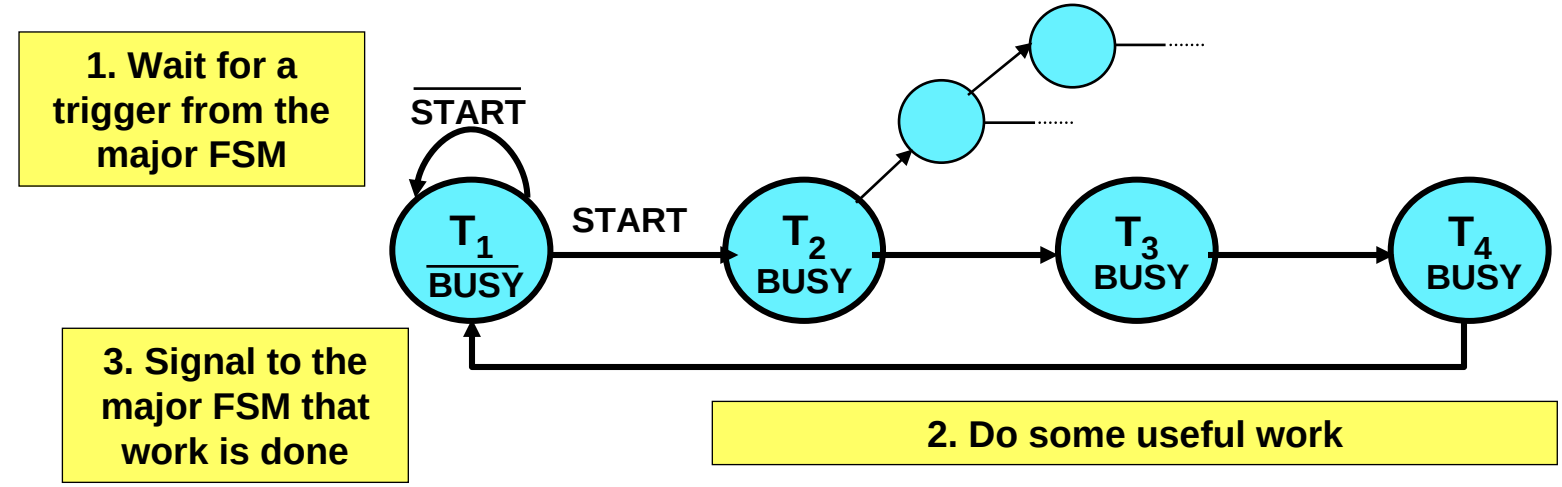
3. Wait until the minor FSM is done

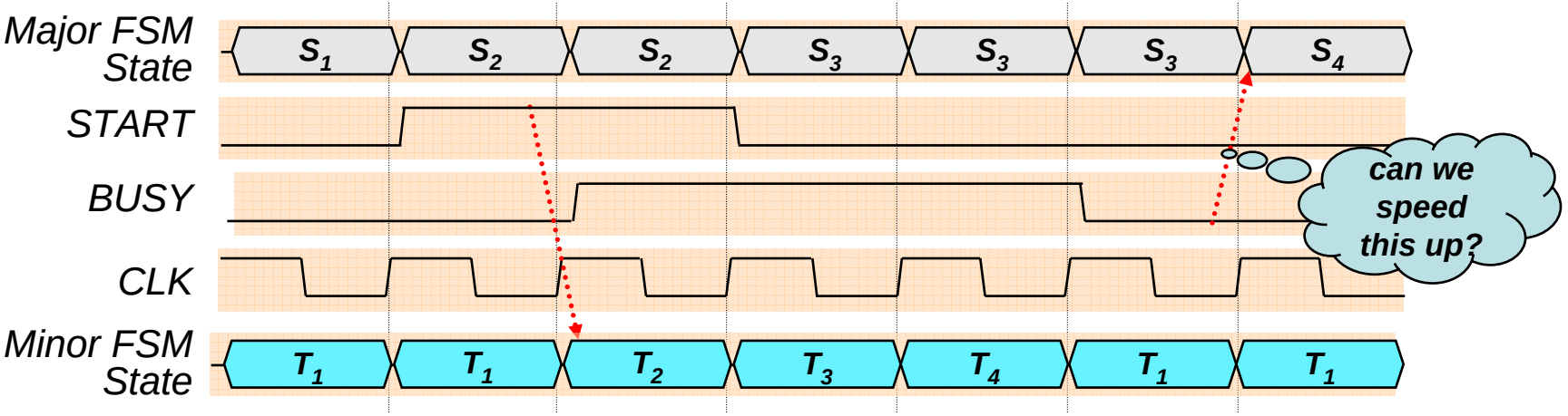


Variations:

- Usually don't need both Step 1 and Step 3
- One cycle "done" signal instead of multi-cycle "busy"

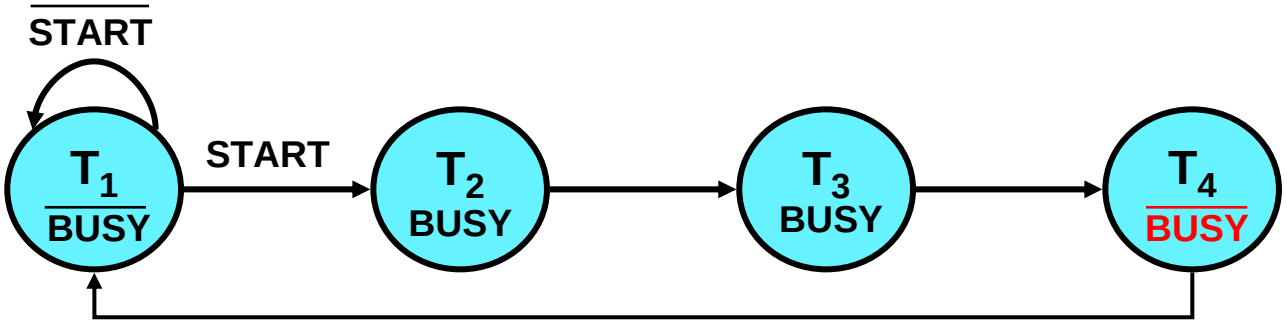
Inside the Minor FSM





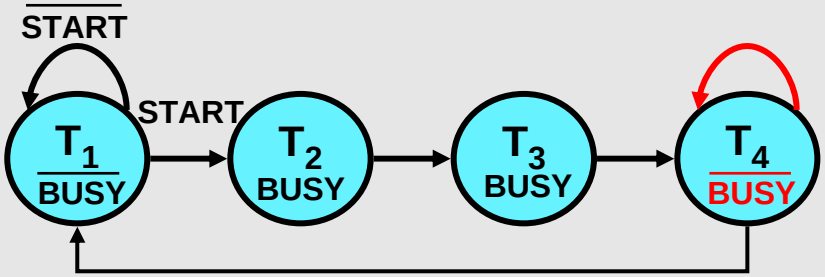
Optimizing the Minor FSM

Good idea: de-assert BUSY one cycle early



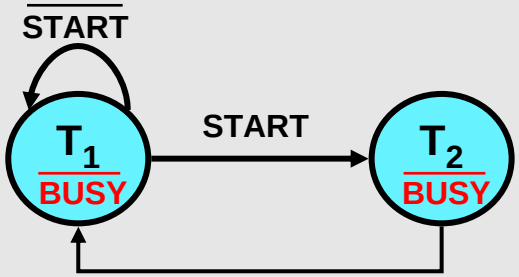
Bad idea #1:

T_4 may not immediately return to T_1

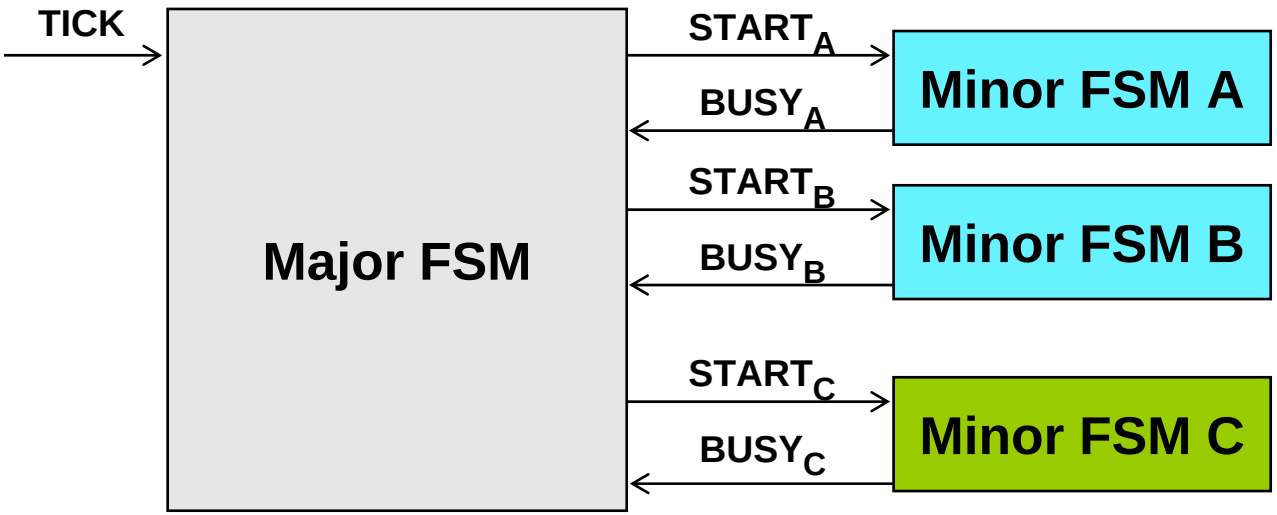


Bad idea #2:

BUSY never asserts!

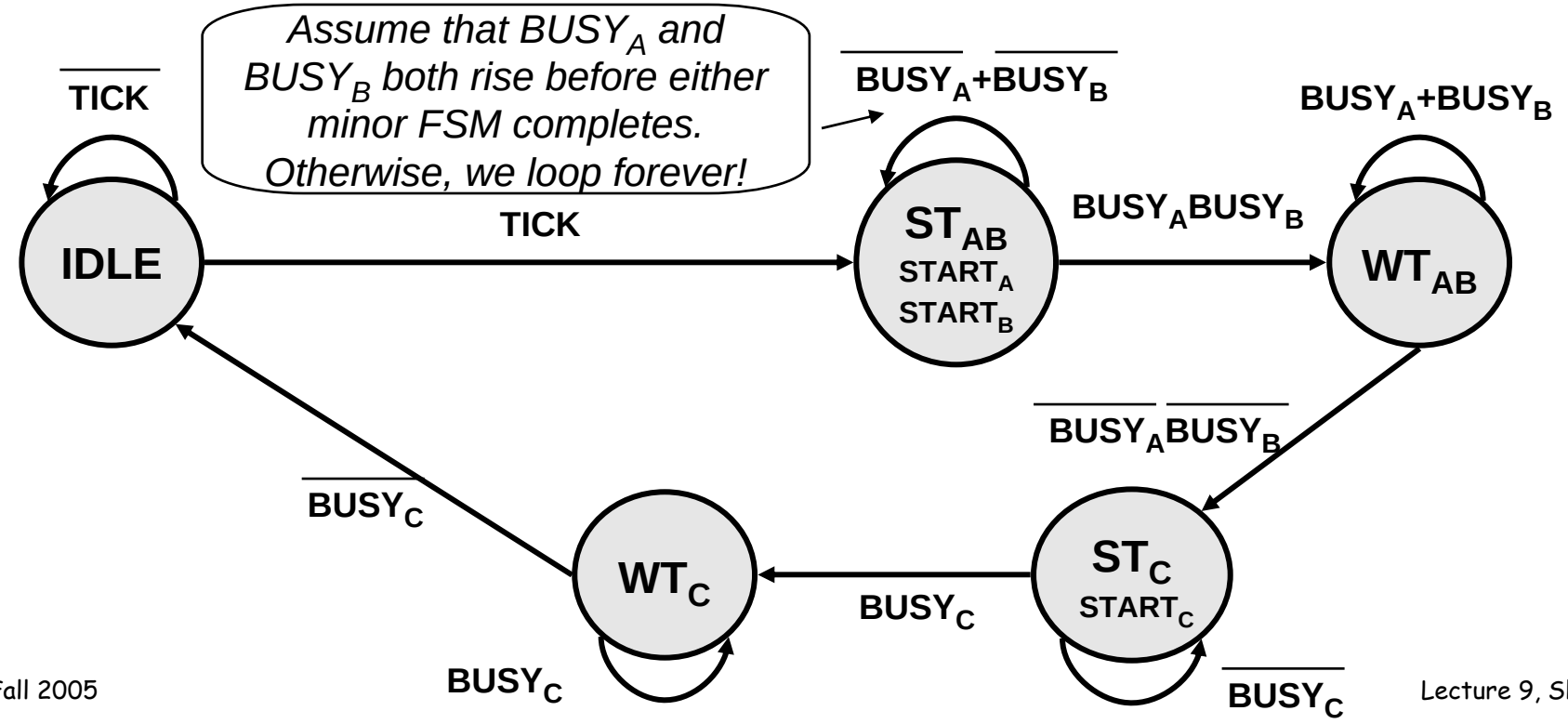


A Four-FSM Example

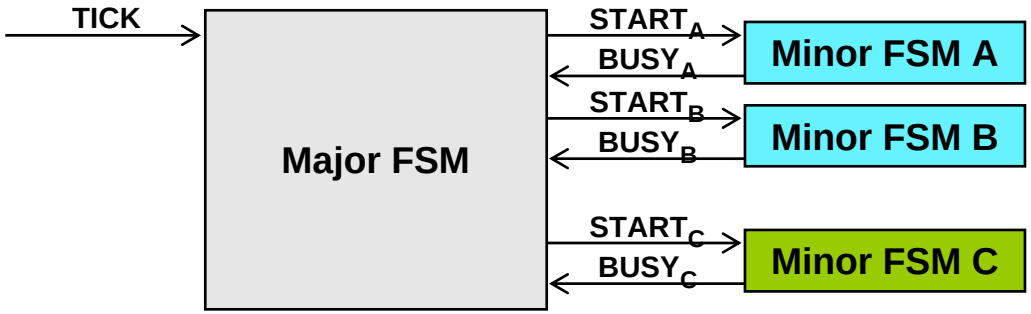


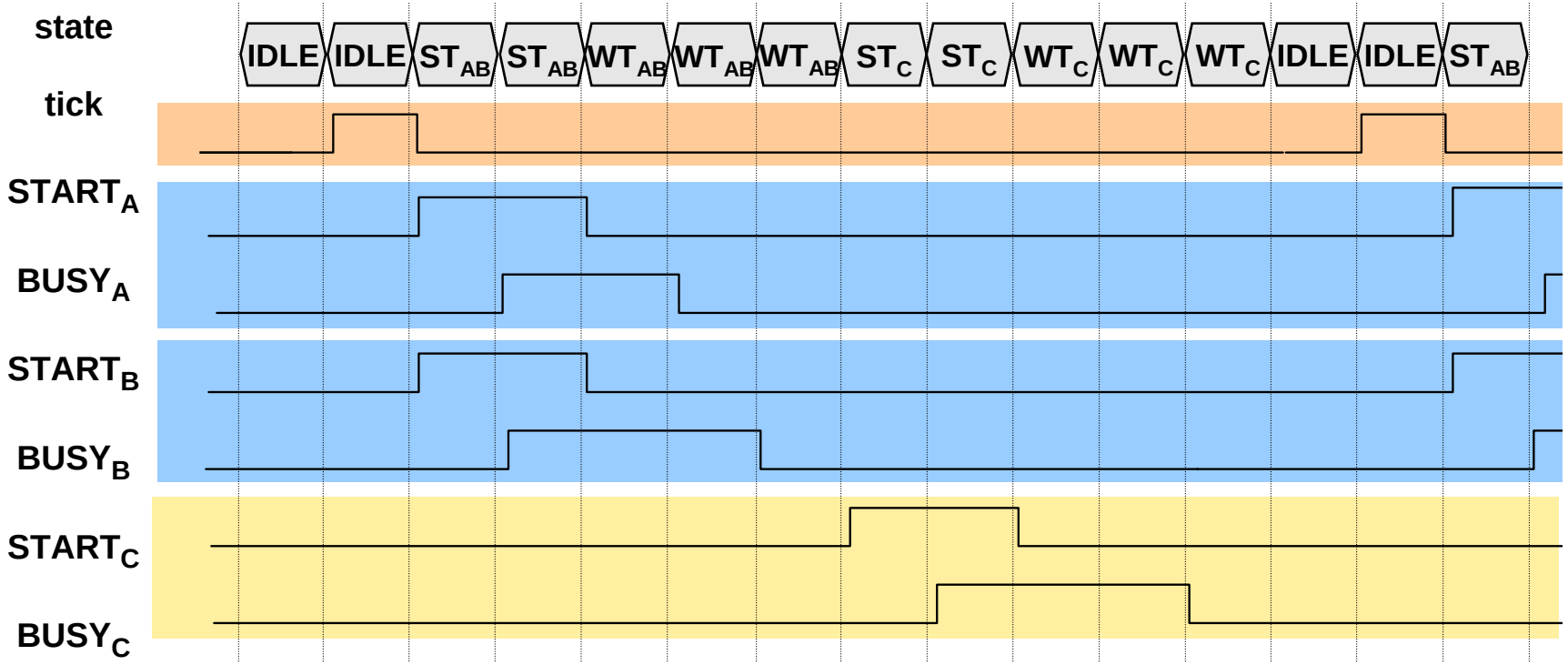
Operating Scenario:

- Major FSM is triggered by TICK
- Minors A and B are started simultaneously
- Minor C is started once both A and B complete
- TICKs arriving before the completion of C are ignored



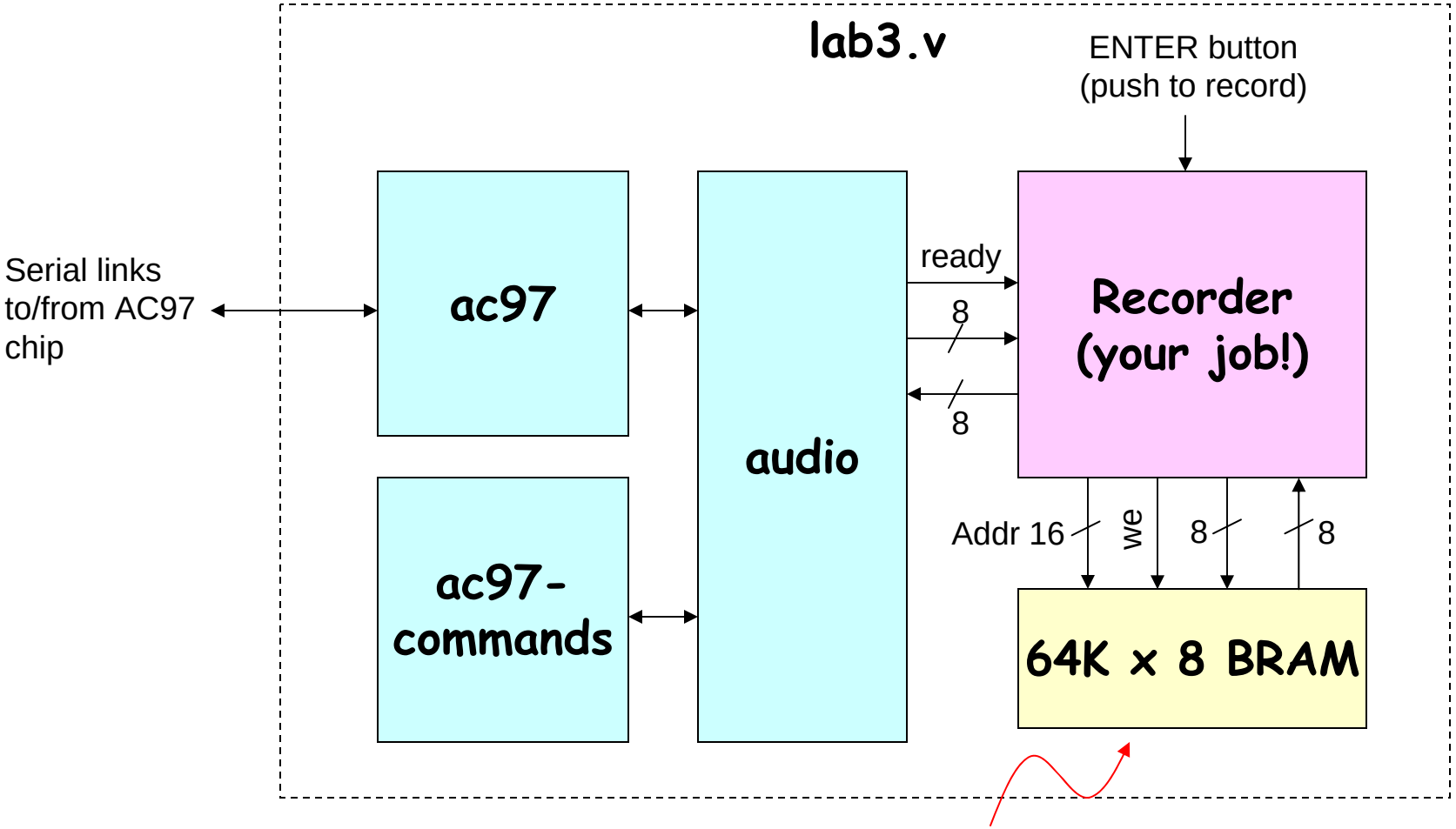
Four-FSM Sample Waveform





Lab 3 overview

Assignment: build a voice recorder that uses 8-bit PCM data @ 6KHz



About 11 seconds of speech @ 6KHz

AC97: PCM data

PCM = pulse code modulation

Sample waveform at fixed intervals, encode results as an N-bit signed number. For our AC97 chip, N = 18.