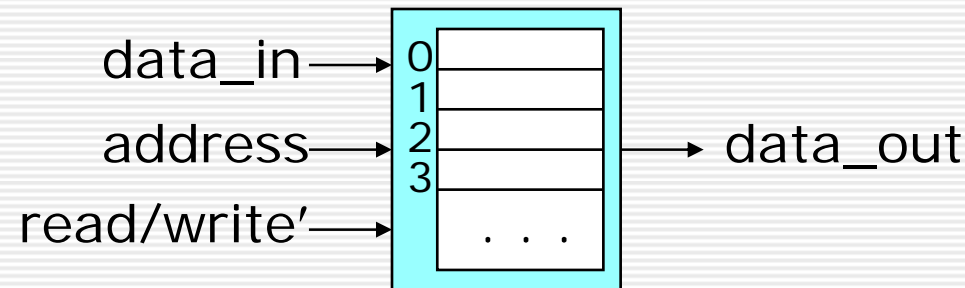


# Introduction to Computer Design

- Memory (W 800-840)
- Basic processor operation
- Processor organization
- Executing instructions
- Processor implementation using VHDL



# Random Access Memory

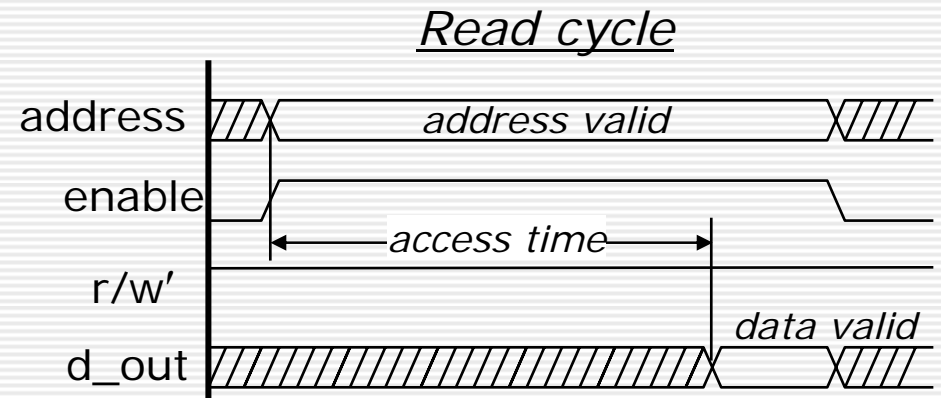


- Logically, a *random access memory* contains an array of numbered storage locations, called *words*.
  - » when read/write' is high, data\_out is equal to the value stored in word specified by address inputs
  - » when read/write' is low, the value on data\_in replaces the value in word specified by address outputs
  - » separate *enable* signal also usually provided
- Simplest RAMs are *asynchronous* - no clock input
  - » synchronous circuits using RAMs must ensure that RAM timing requirements are satisfied to ensure correct operation

# Timing of RAM Operations

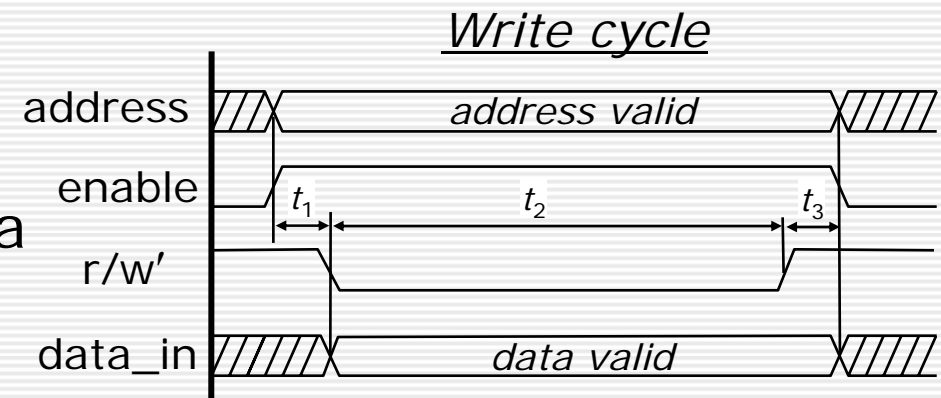
## ■ Read cycle

- » *access time*: time from “last” address change until output data is valid



## ■ Write cycle

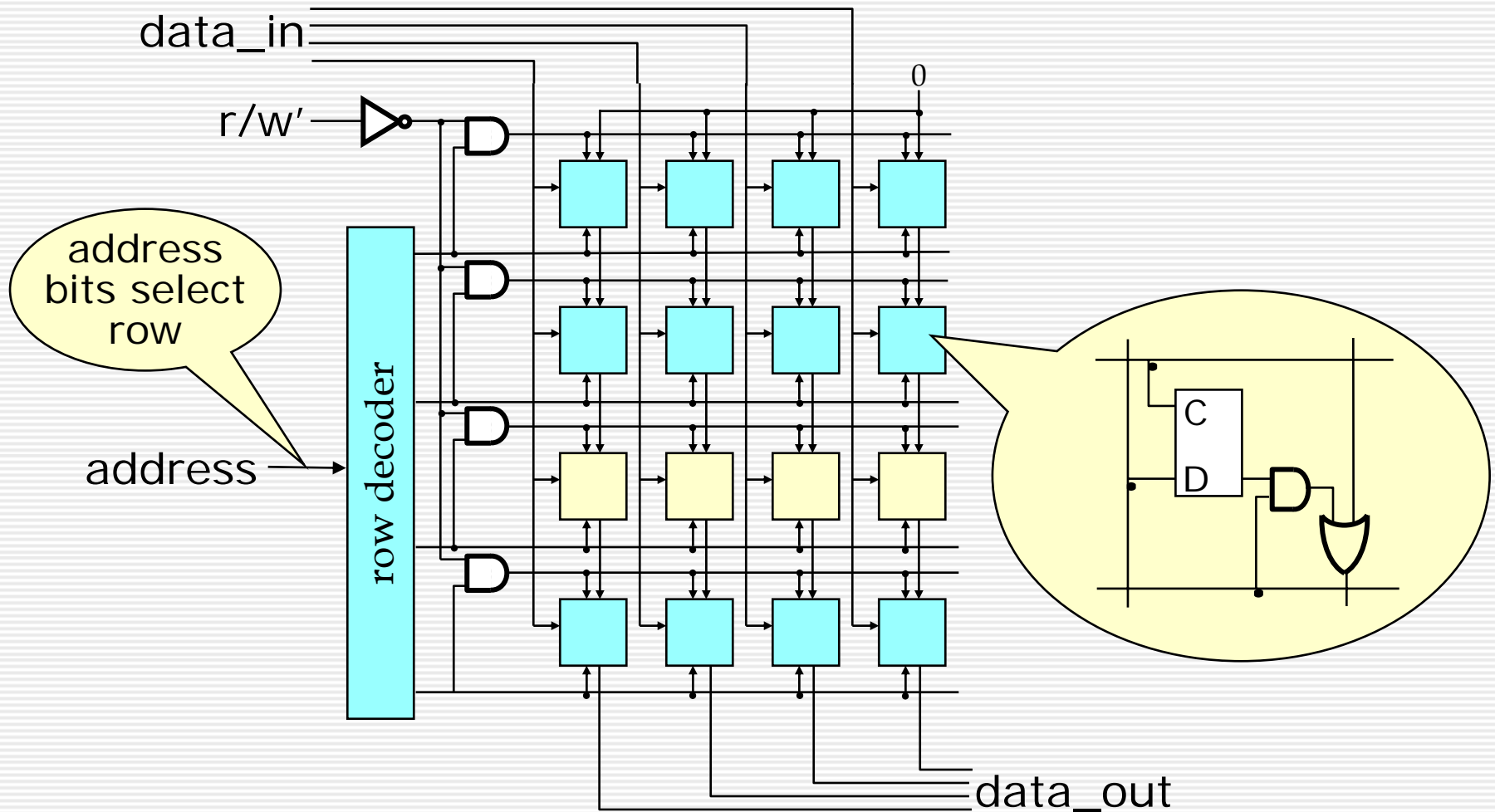
- »  $t_1$  is min time from address stable and enable asserted until r/w' is lowered
- »  $t_2$  is min time that input data must remain stable before r/w' can be raised
- »  $t_3$  is min time that address stays valid after r/w' is raised
- » *cycle time* is  $t_1 + t_2 + t_3$



- Circuits using RAM must ensure timing conditions are met.

# Implementing a RAM

4 word RAM with 4 bits per word



# VHDL Model for SRAM

```
entity lram16x64 is port (  
    reset, en, r_w: in STD_LOGIC;  
    aBus: in STD_LOGIC_VECTOR(adrLength-1 downto 0);  
    dBus: inout STD_LOGIC_VECTOR(wordSize-1 downto 0));  
end lram16x64;  
architecture arch of lram16x64 is  
type ram_typ is array(0 to memSize-1)  
    of STD_LOGIC_VECTOR(wordSize-1 downto 0);  
signal ram: ram_typ;  
begin  
    process(reset, en, r_w, aBus, dBus) begin  
        if reset = '1' then  
            ram(0)  <= x"aaaa";  
            ram(1)  <= x"5555";  
        elsif en = '1' and r_w = '0' then  
            ram(int(aBus)) <= dBus;  
        end if;  
    end process;  
    dBus <= ram(int(aBus))  
        when reset = '0' and en = '1' and r_w = '1' else  
        (dbus'range => 'Z');  
end arch;
```

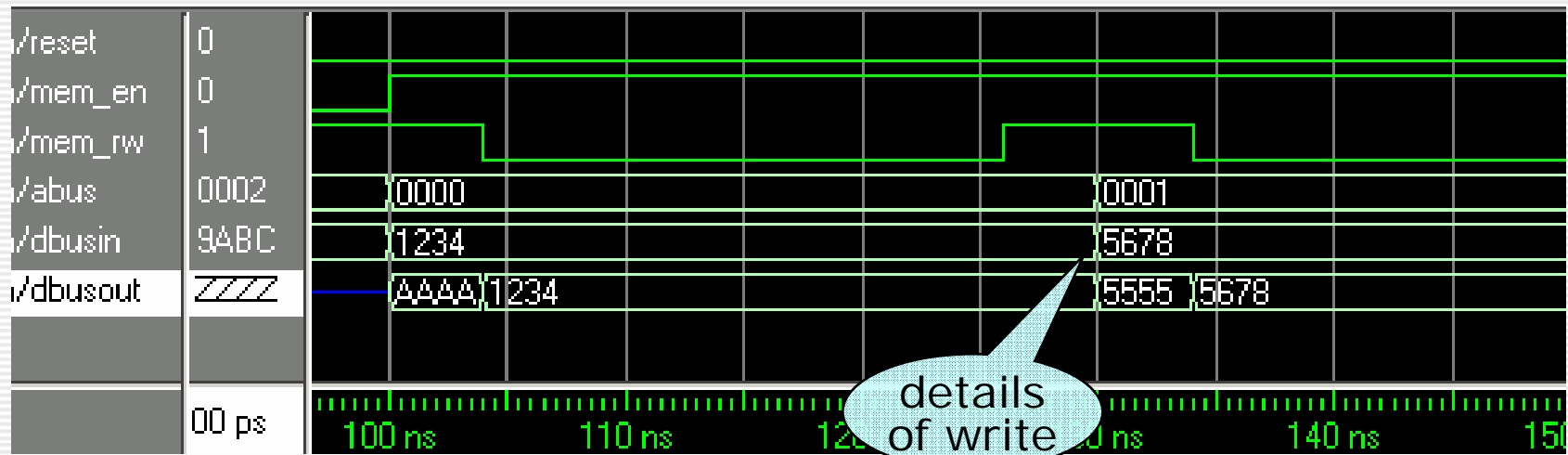
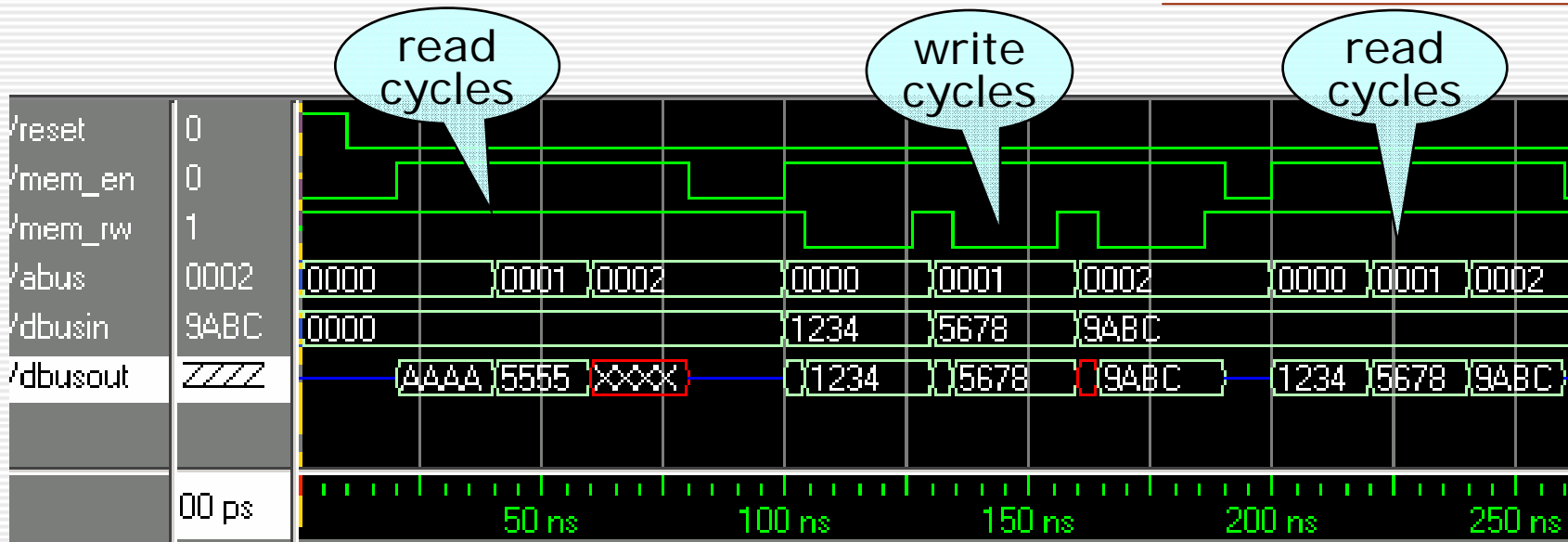
array of 16  
bit words for  
memory

initialize  
memory

write to location  
specified by  
address bus

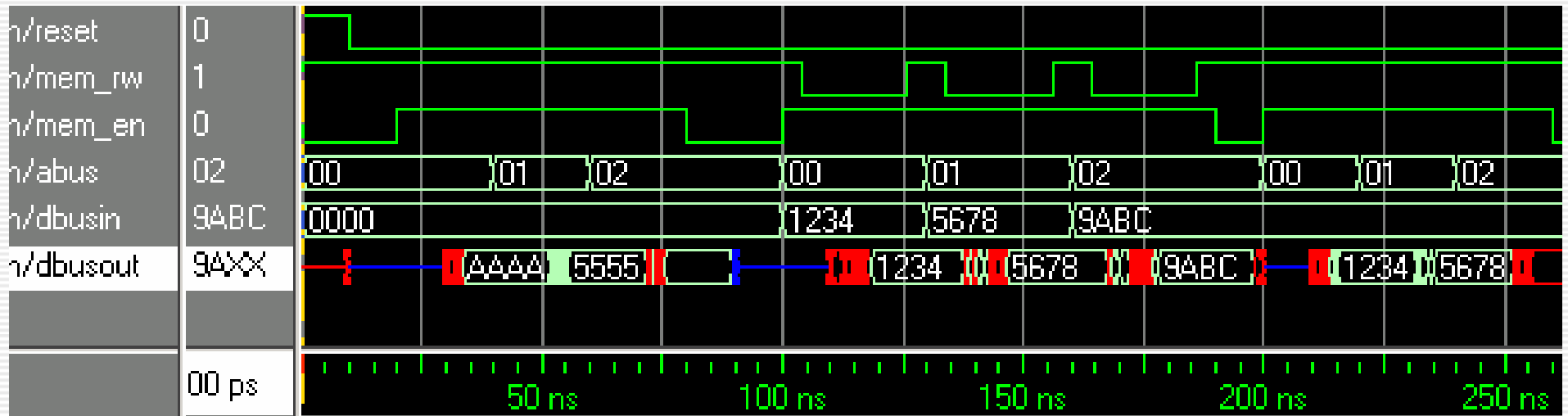
read from  
location specified  
by address bus

# Functional Simulation

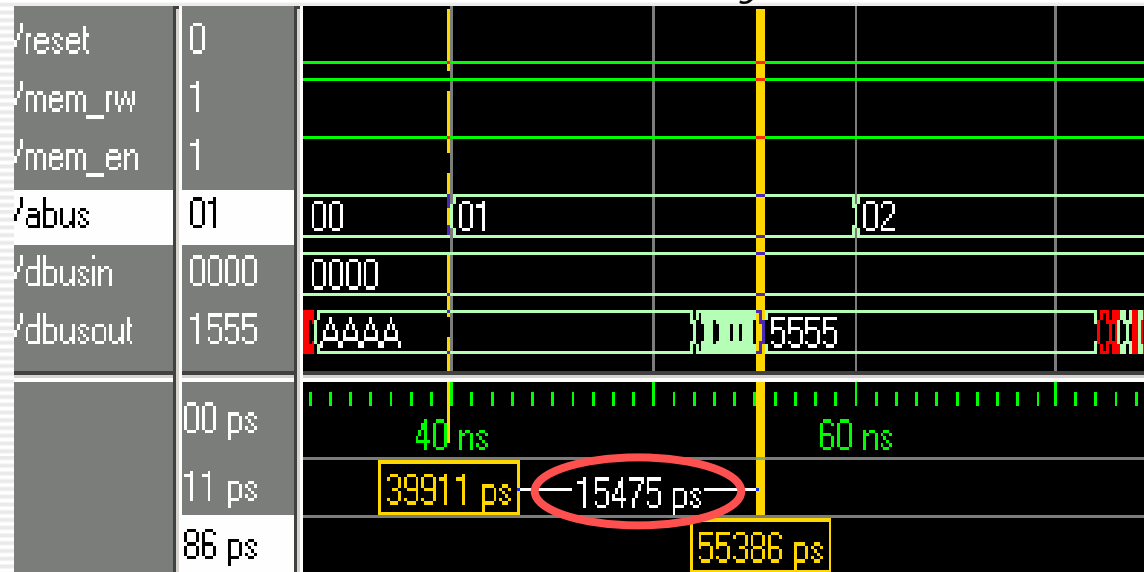


# Timing Simulation

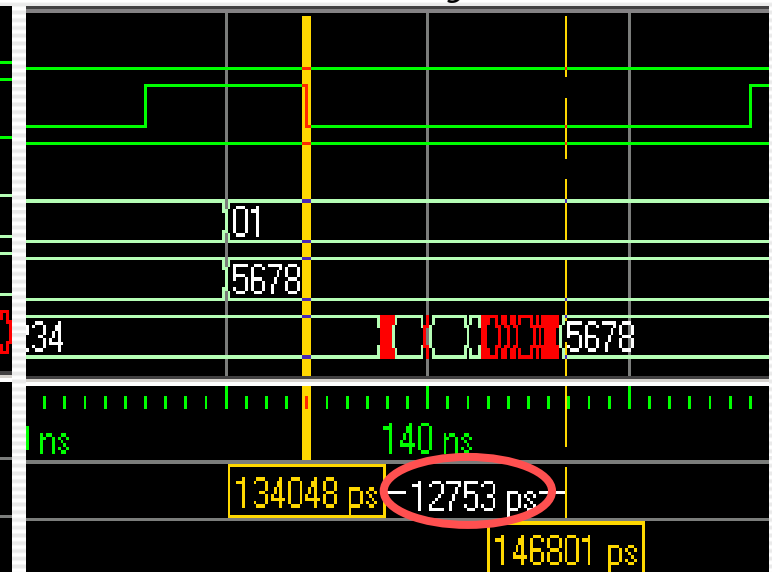
## Overview



## Read Cycle Detail

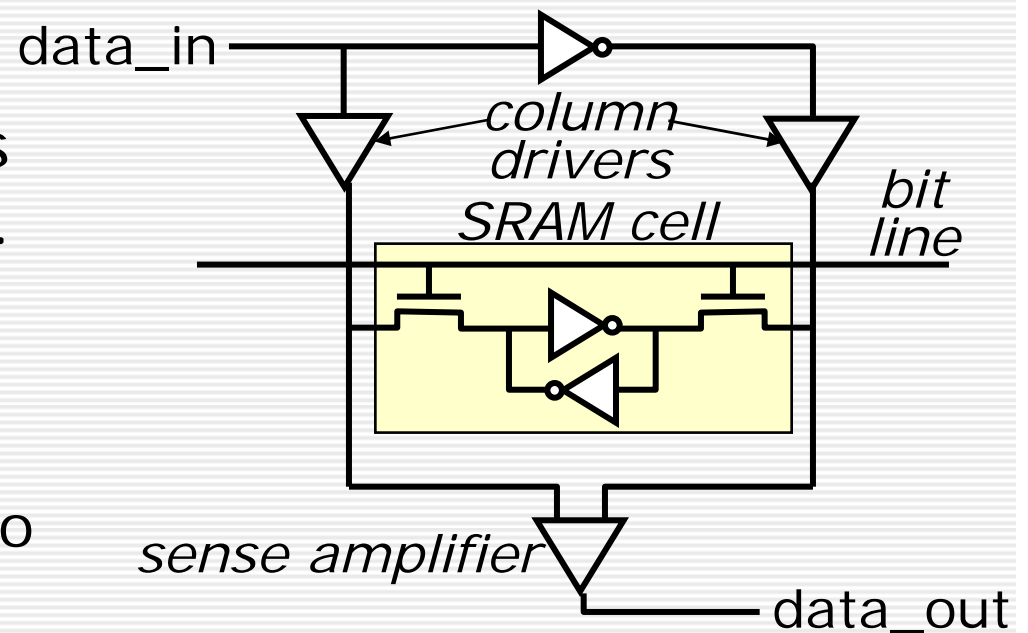


## Write Cycle Detail

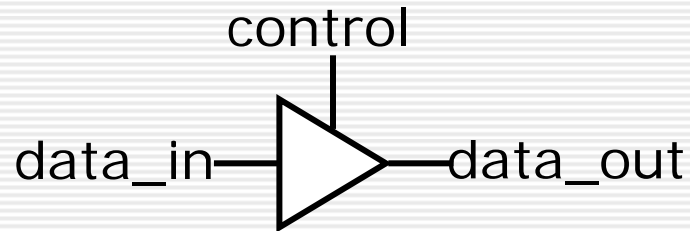


# Shrinking the Memory Cell

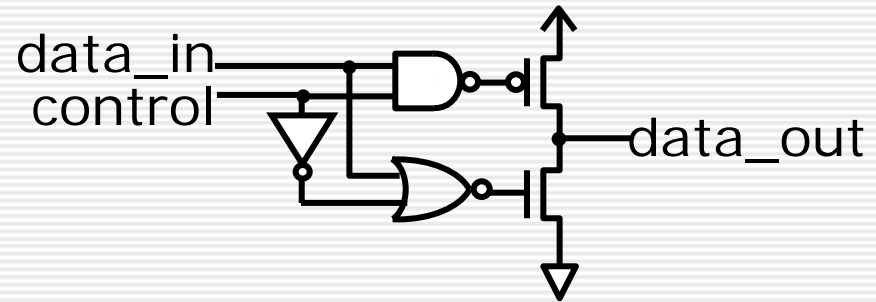
- Computers and other digital systems generally use large amounts of memory.
- Specialized memory cells have been developed to pack more memory in given amount of space.
- Typical *static RAM* uses 6 transistor cell using pair of inverters and pair of pass transistors.
- *Bit line* asserted to read or write the cell.
- Complementary data lines used for input and output.
- Column drivers enabled when writing.
  - » large current capacity allows them to force cell to desired state



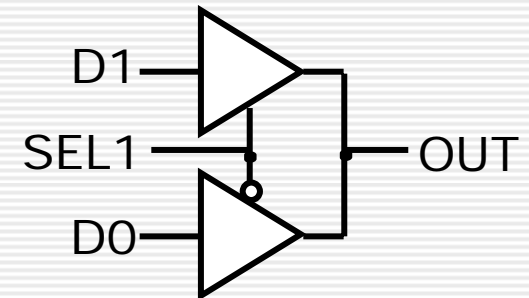
# Tri-State Buffers



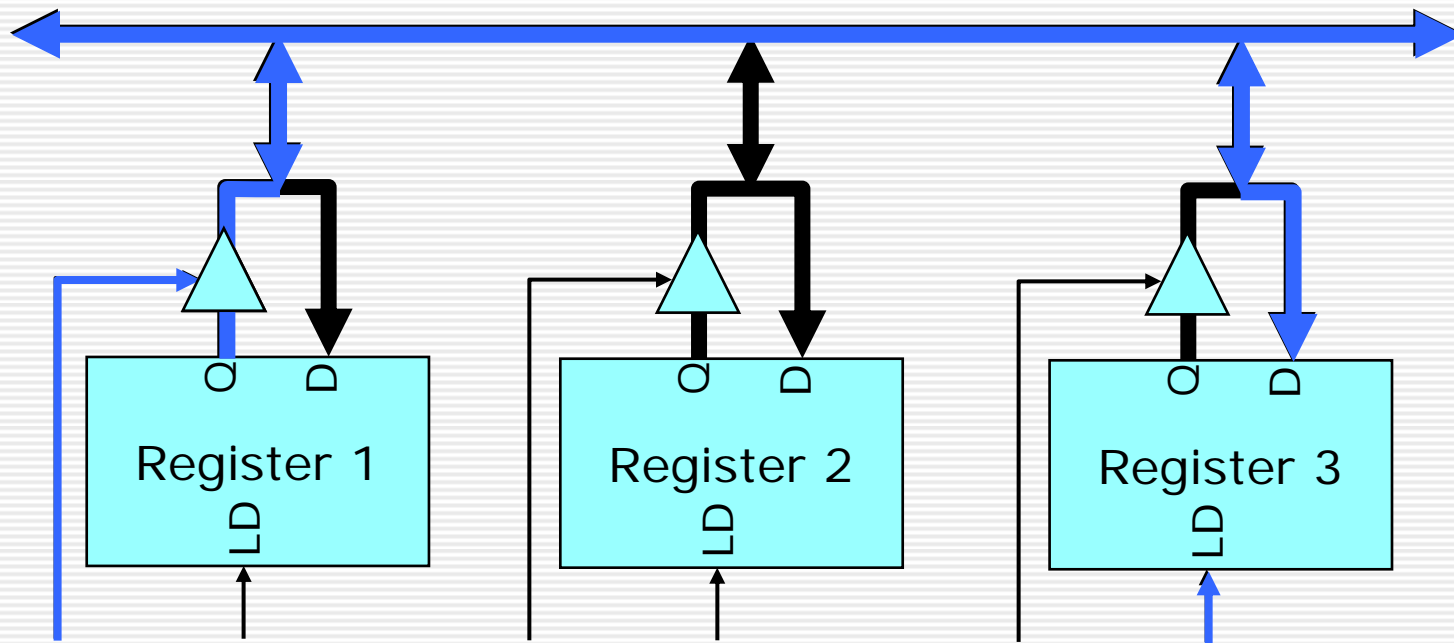
## CMOS implementation



- A *tri-state buffer* has a data input and a control input.
  - » when control input is asserted, output equals input
  - » when control input is not asserted, output is disconnected from input - called *high impedance state*
- Tri-state buffers, can be used to build "distributed" multiplexors.
- Shared outputs are called *buses*.
- Also allows single wire to be used as data input and output.

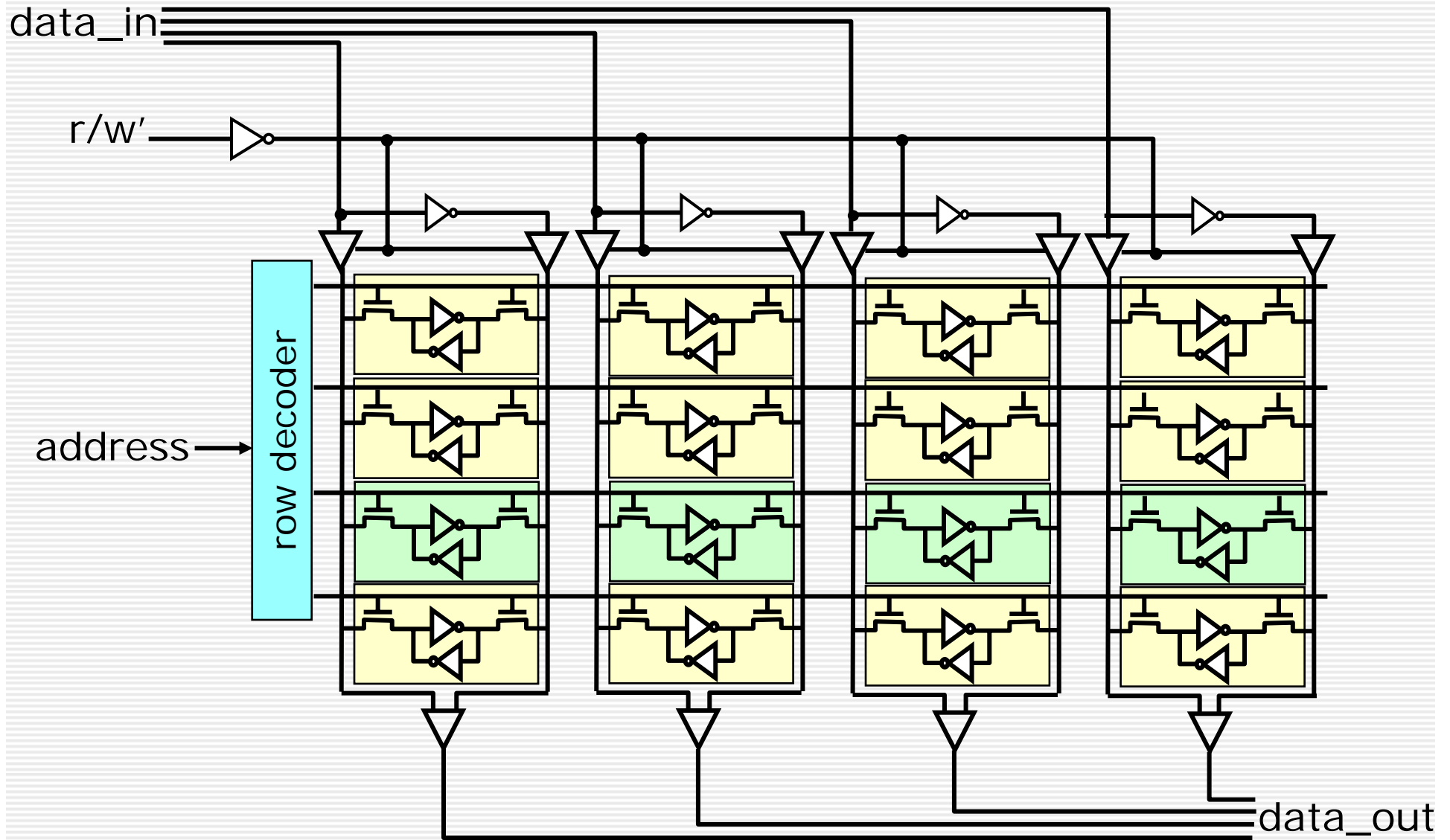


# Data Transfer Using Buses

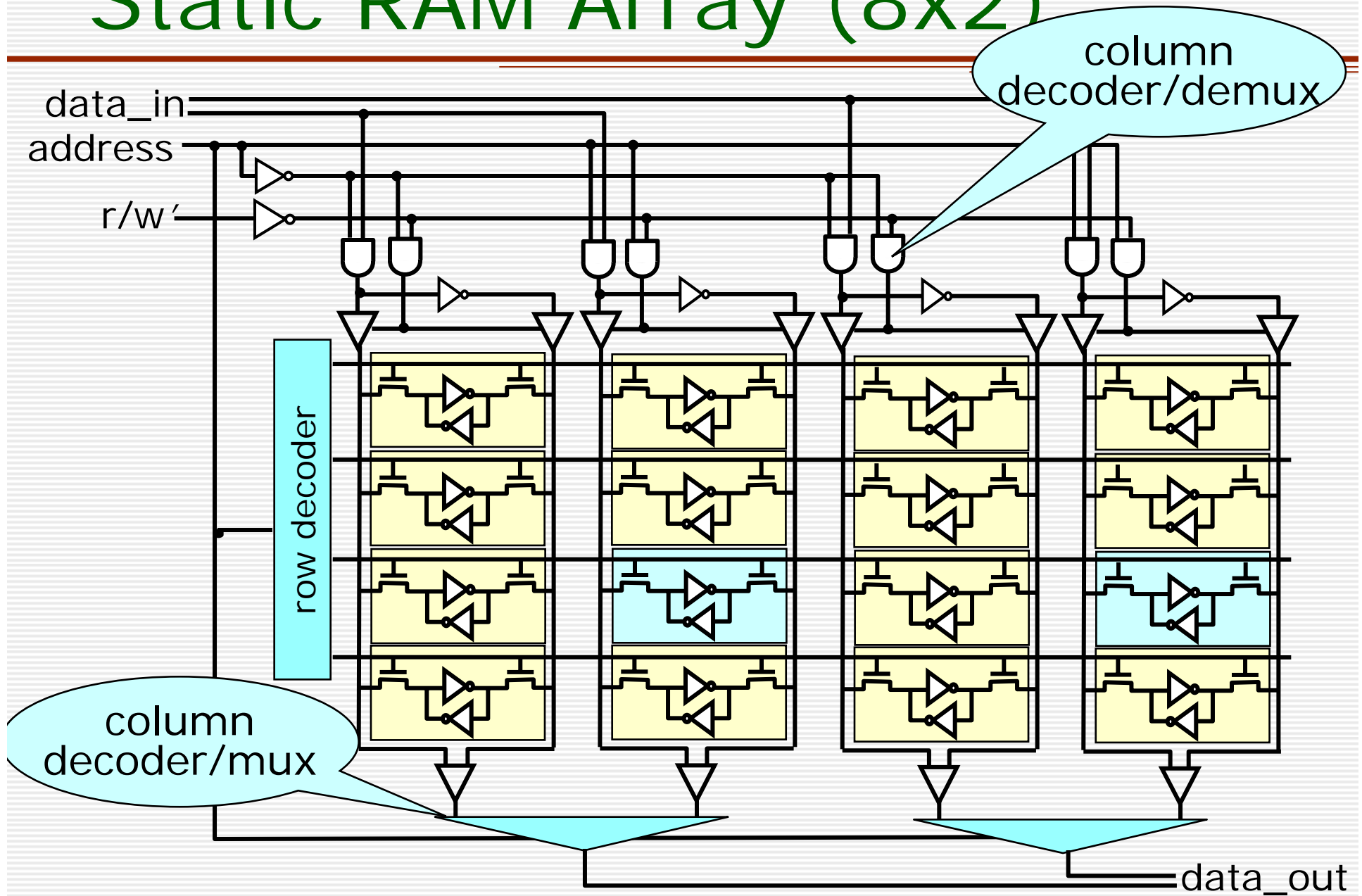


- A *bus* is a shared set of wires used to transfer data among any of several sources/destinations.
- Data transfers involve:
  - » enabling source to place data on the bus
  - » loading data into destination

# Static RAM Array (4x4)

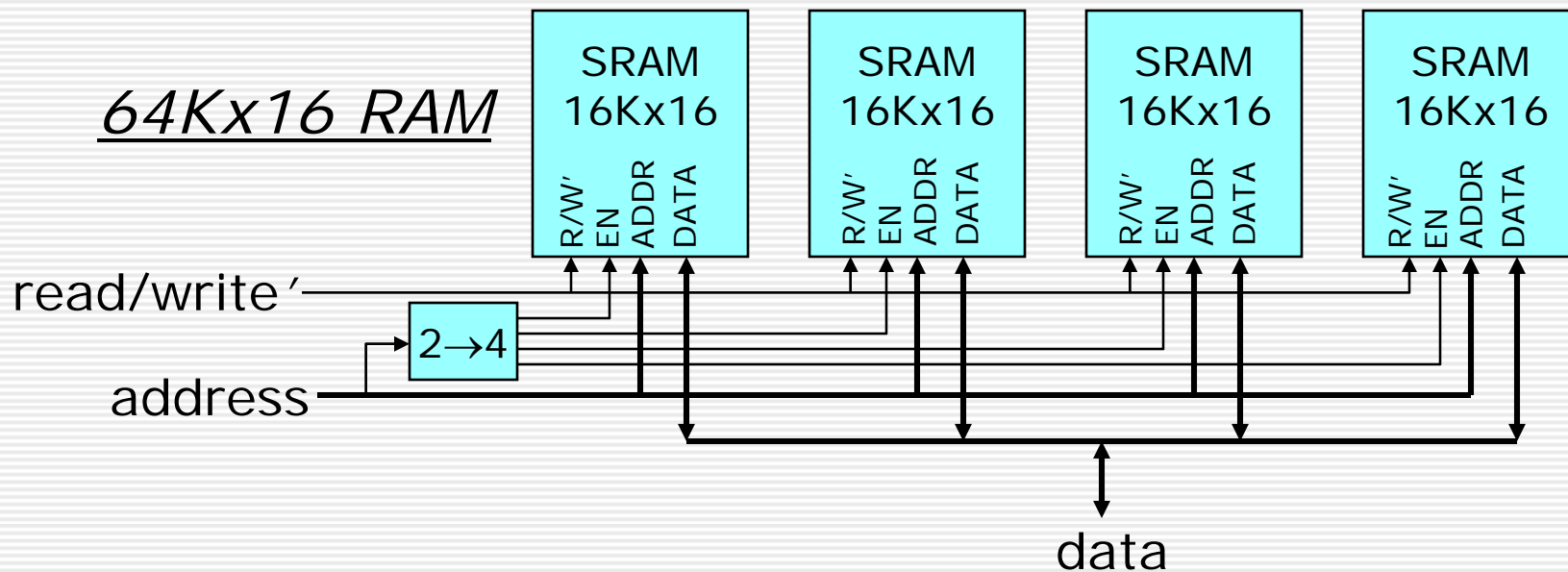


# Static RAM Array (8x2)



# Building Larger RAMs

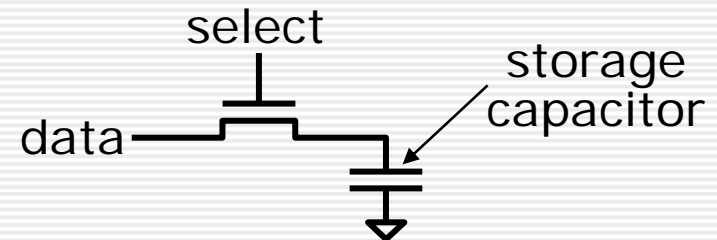
- Systems often require larger RAMs than can be constructed using a single SRAM component.
- The use of an external decoder and the enable input allows larger RAMs to be constructed.



- Alternative design uses 64Kx4 RAM chips.
  - » no external decoder needed in this case

# Dynamic RAMs

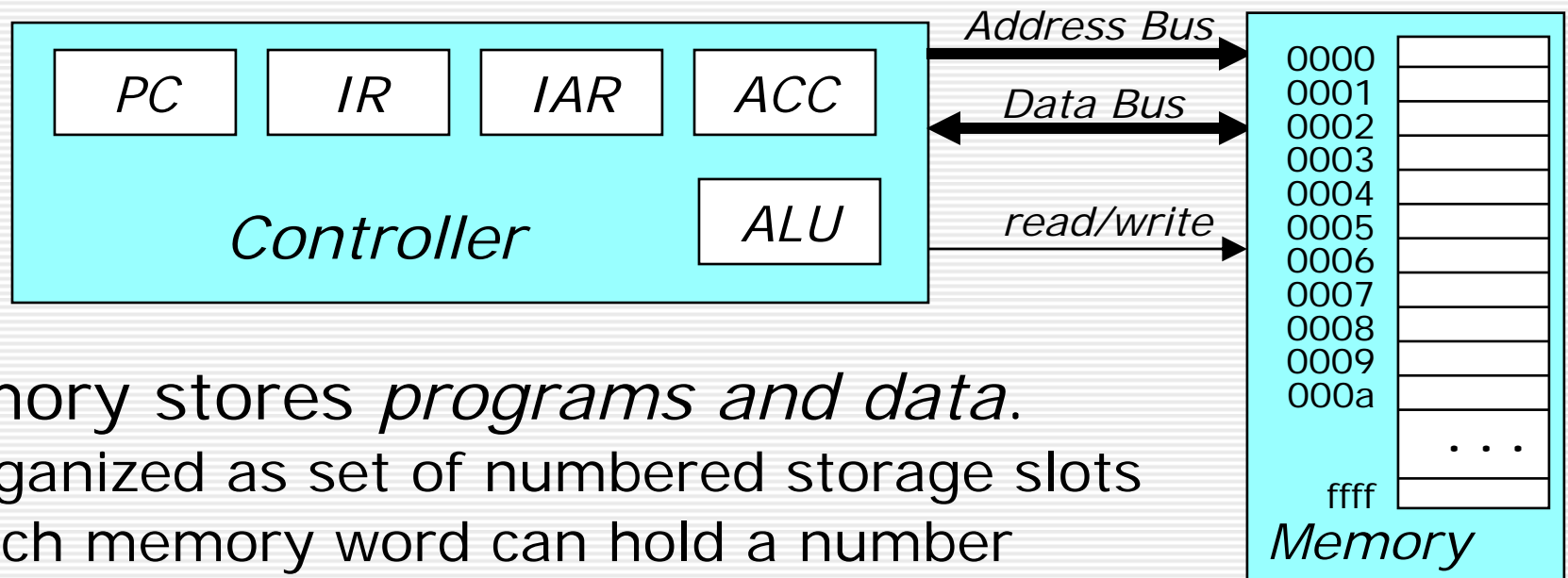
- Dynamic RAMs use a simpler memory cell to enable more bits to be stored in a single chip (4-8x).
  - » each storage cell consists of a pass transistor and a capacitor
  - » reading contents, destroys value
    - need to write back after reading
  - » stored charge *leaks* from capacitor after 10-100 ms
    - requires periodic *refresh* of memory contents
- DRAM cells are organized in 2D arrays, much like those for SRAM.
  - » single data line rather than pair
  - » requires sensitive *sense amplifiers* to detect stored charge
  - » takes more time (10x) to read values than with SRAM



# Addressing in Dynamic RAMs

- Large memory chips require lots of address pins.
- Many DRAM chips reduce number of address pins by dividing address into two parts.
  - » *row address* determines which row in 2D array is selected
  - » *column address* selects one or more bits in the row
- Column address can be provided after row address without slowing down memory access.
  - » so, same address pins can be used to supply both row and column addresses
  - » *Row Address Strobe (RAS)*, *Column Address Strobe (CAS)* used to load row and column addresses into on-chip registers
- Refresh circuitry periodically reads each row in memory array and writes it back – often built into chip.

# Basic Processor & Memory



- Memory stores *programs and data*.
  - » organized as set of numbered storage slots
  - » each memory word can hold a number
  - » processor can read from or write to any word
- Fetch & execute cycle
  - » read word whose address is in *Program Counter* (PC) and increment PC
  - » interpret stored value as *instruction* (decoding)
  - » perform instruction using *Accumulator* (ACC) and *Arithmetic & Logic Unit* (ALU)

# Instruction Set

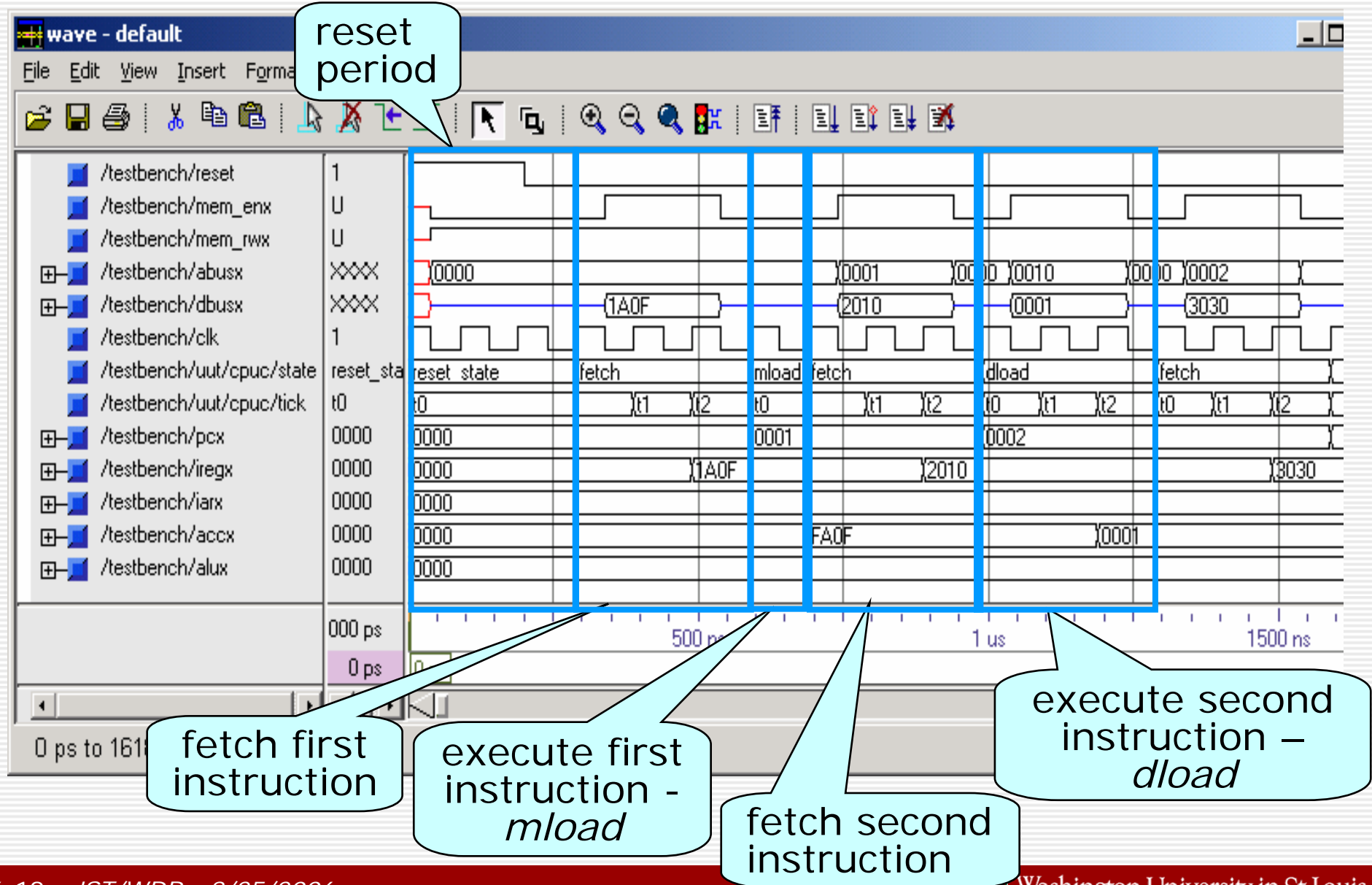
0000	halt - halt execution
0001	negate - $ACC := -ACC$
1xxx	immediate load – if sign bit of xxx is 0 then $ACC := 0xxx$ else $ACC := fxxx$
2xxx	direct load - $ACC := M[0xxx]$
3xxx	indirect load - $ACC := M[M[0xxx]]$
4xxx	direct store - $M[0xxx] := ACC$
5xxx	indirect store – $M[M[0xxx]] := ACC$
6xxx	branch - $PC := 0xxx$
7xxx	branch if zero - if $ACC = 0$ then $PC := 0xxx$
8xxx	branch if positive - if $ACC > 0$ then $PC := 0xxx$
9xxx	branch if negative - if $ACC < 0$ then $PC := 0xxx$
axxx	add - $ACC := ACC + M[0xxx]$

# Simple Program

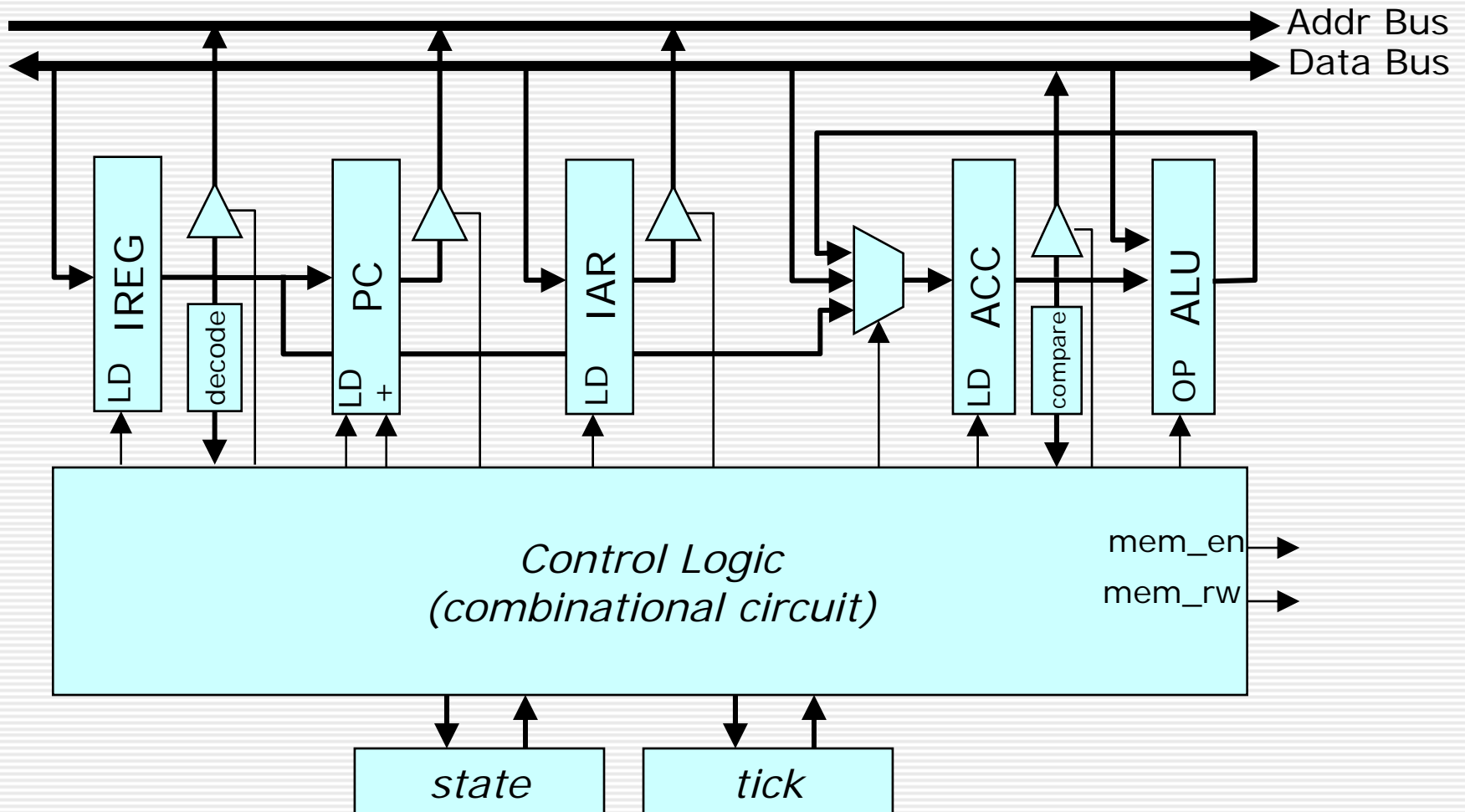
- Add the values in locations 20-2f and write sum in 10.

<u>Address</u>	<u>Instruction</u>	<u>Comment</u>
0000 (start)	1000 (ACC := 0000)	initialize sum
0001	4010 (M[0010] := ACC)	
0002	1020 (ACC := 0020)	initialize pointer
0003	4011 (M[0011] := ACC)	
0004 (loop)	1030 (ACC := 0030)	if pointer = 030, quit
0005	0001 (ACC := -ACC)	
0006	a011 (ACC := ACC + M[0011])	
0007	700f (if 0 goto 000f)	
0008	3011 (ACC := M[M[0011]])	sum = sum + *pointer
0009	a010 (ACC := ACC + M[0010])	
000a	4010 (M[0010] := ACC)	
000b	1001 (ACC := 0001)	pointer = pointer + 1
000c	a011 (ACC := ACC + M[0011])	
000d	4011 (M[011] := ACC)	
000e	6004 (goto 0004)	goto loop
000f (end)	0000 (halt)	halt
0010		Store sum here
0011		Pointer to next value

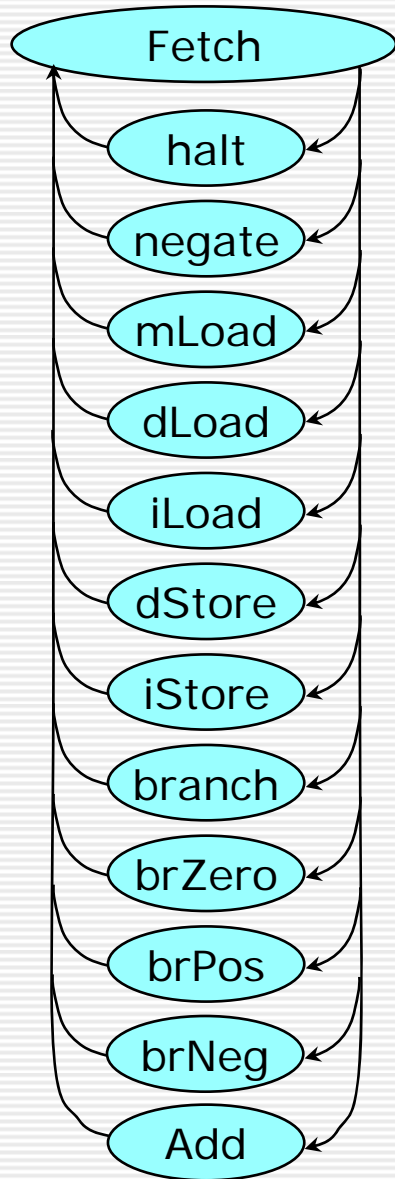
# Execution of a Computer Program



# Detailed Processor Diagram



# Processing Cycle



- Instruction fetch
  - » *PC* used to read word from memory
  - » *PC* is incremented
- Instruction decode
  - » first 4 bits of retrieved instruction are decoded to determine what to do
  - » appropriate circuitry activated
- Instruction execution
  - » retrieve additional memory words
  - » write to memory
  - » modify *PC* or *ACC* contents
  - » may take different amounts of time to complete

# Instruction Execution

## ■ Direct Load

- » transfer data from memory to *ACC*, using low 12 bits of instruction word as memory address
- » requires asserting of memory signals and loading *ACC*

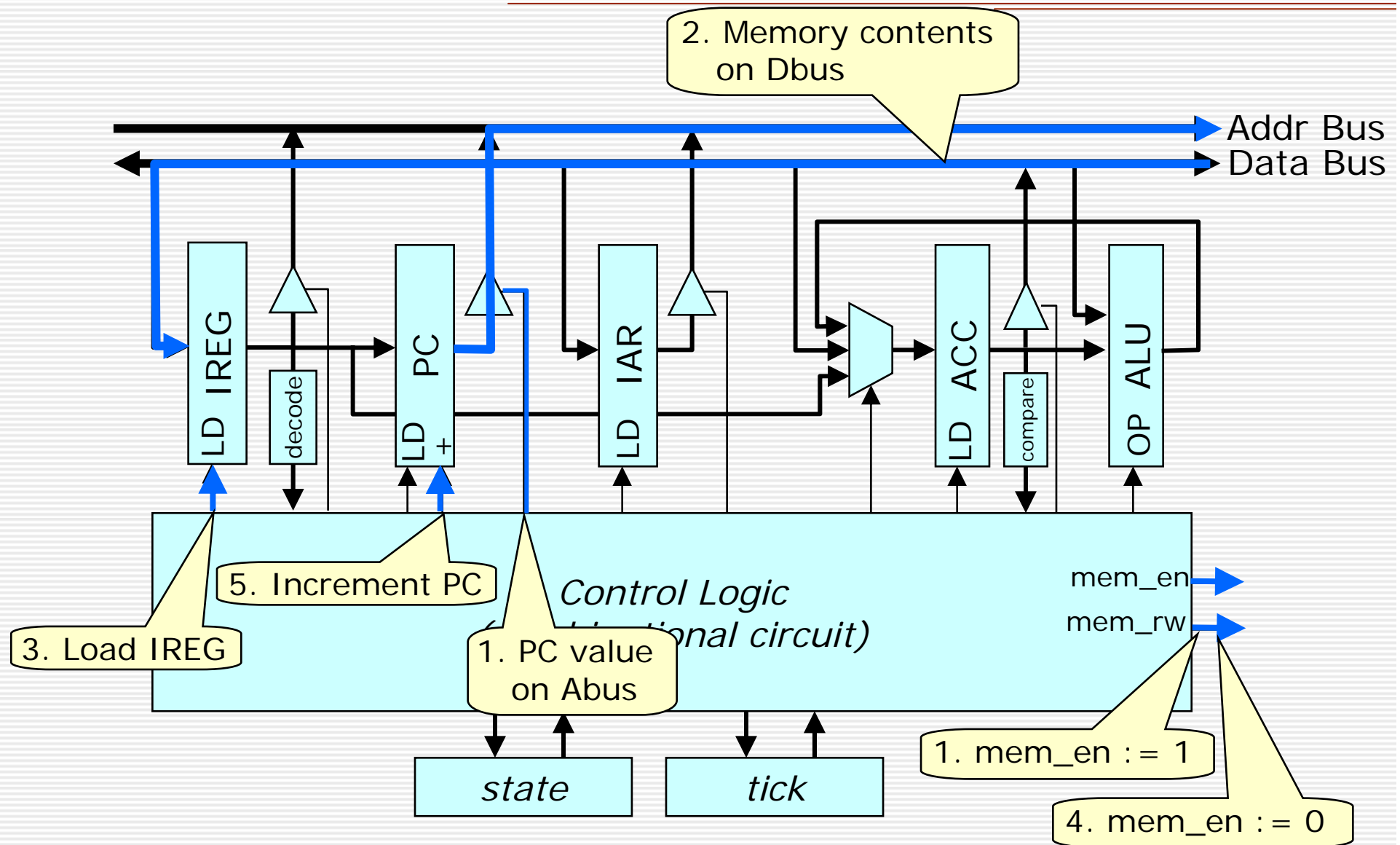
## ■ Conditional branch

- » determine if  $ACC=0$  (or  $>0$  or  $<0$ )
- » if so, transfer low 12 bits of instruction word to *PC*

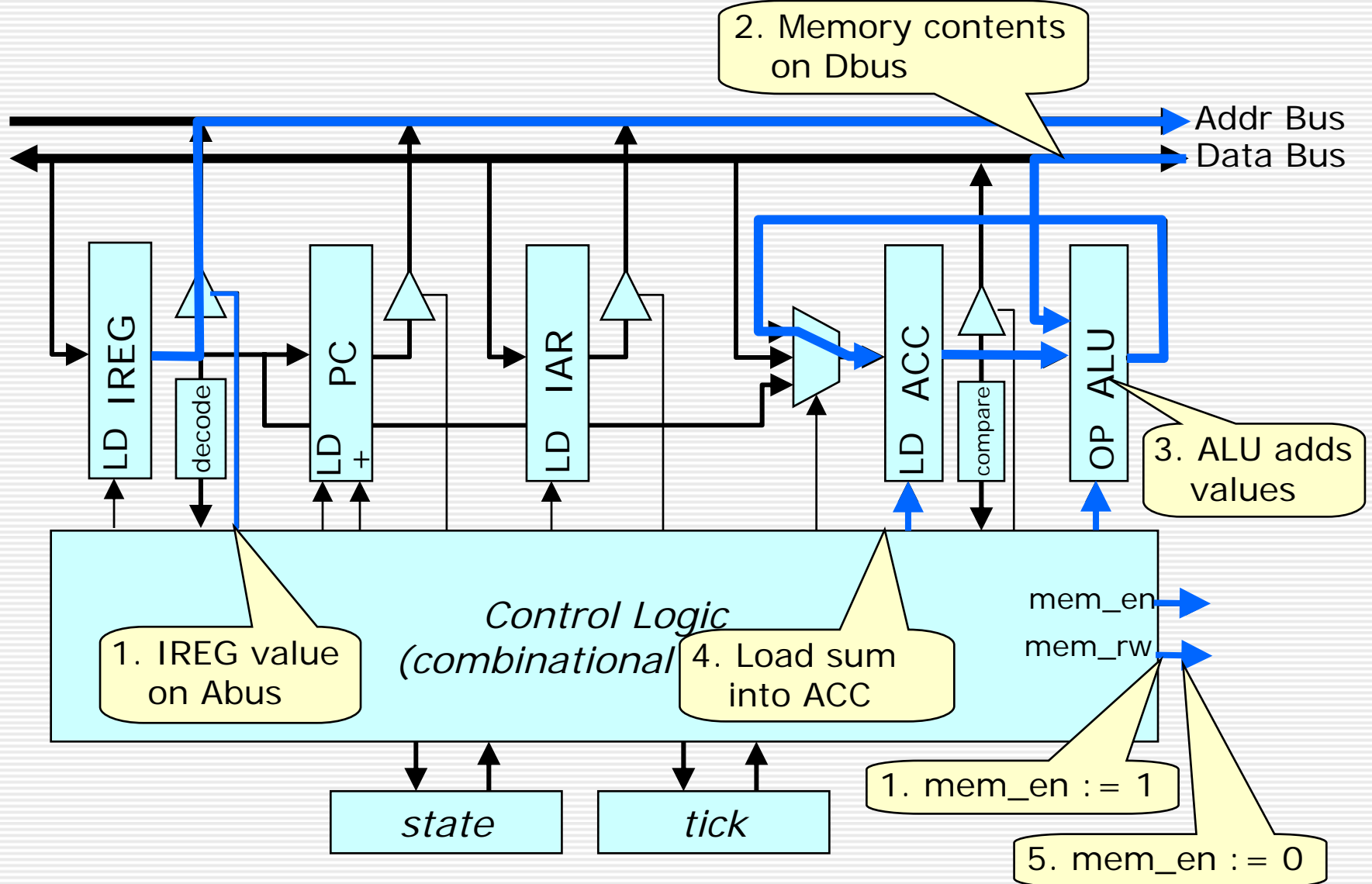
## ■ Indirect store

- » transfer data from memory to Indirect Address Register (*IAR*) using low 12 bits of instruction word as memory address
- » transfer data from *ACC* to memory, using *IAR* contents as address
- » requires placing *IAR* value on address bus and asserting signals to perform memory write

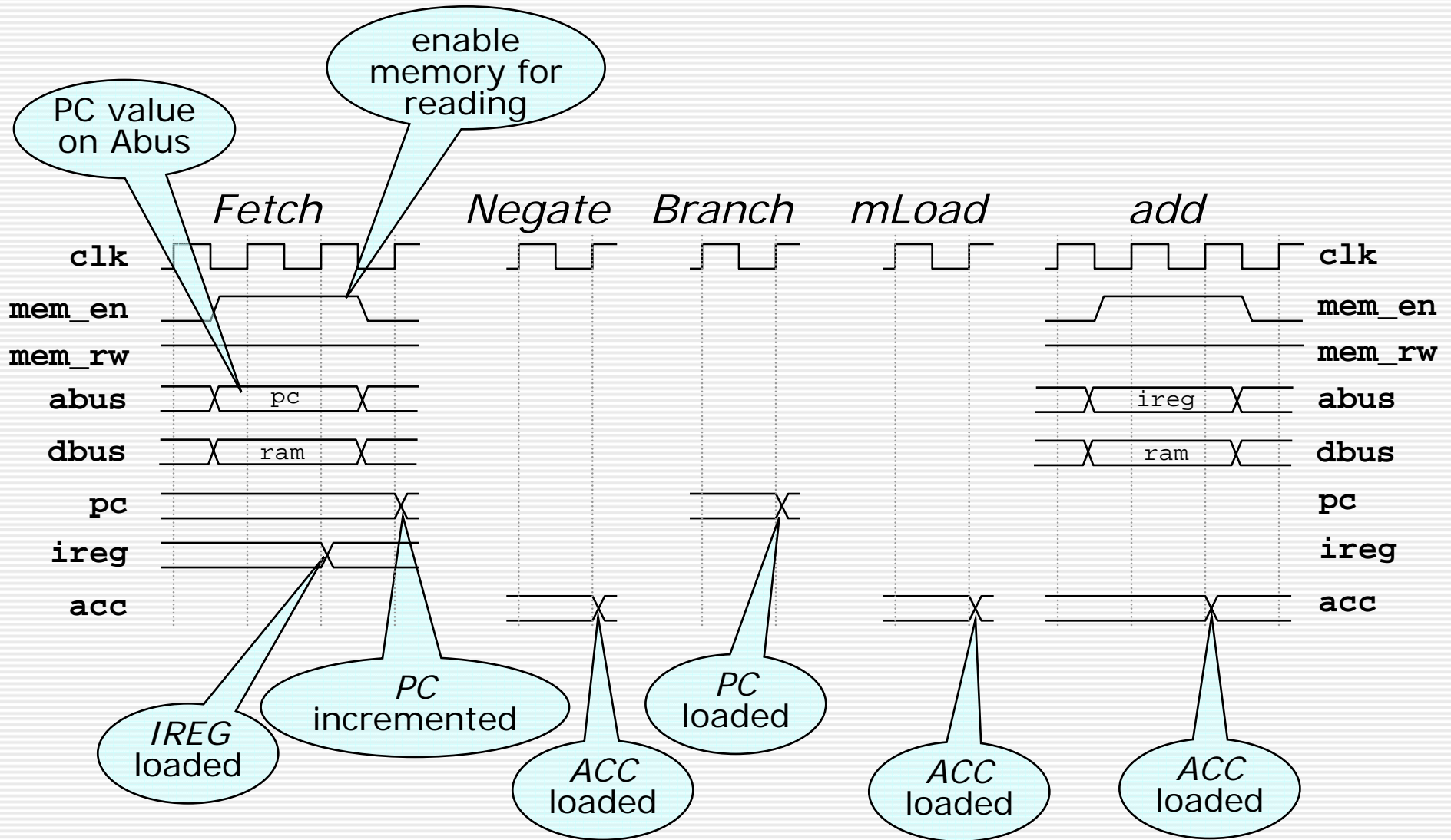
# Instruction Fetch



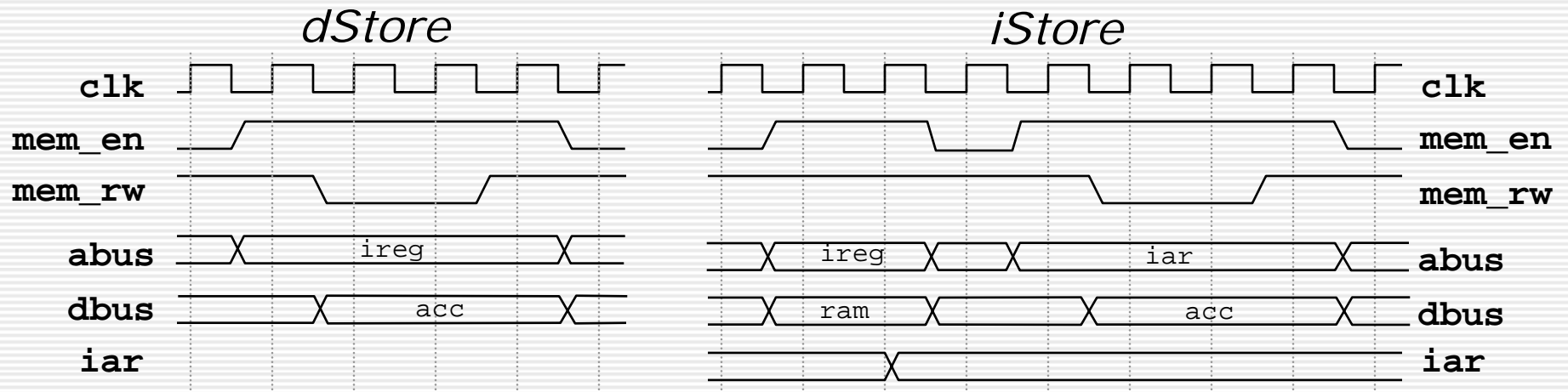
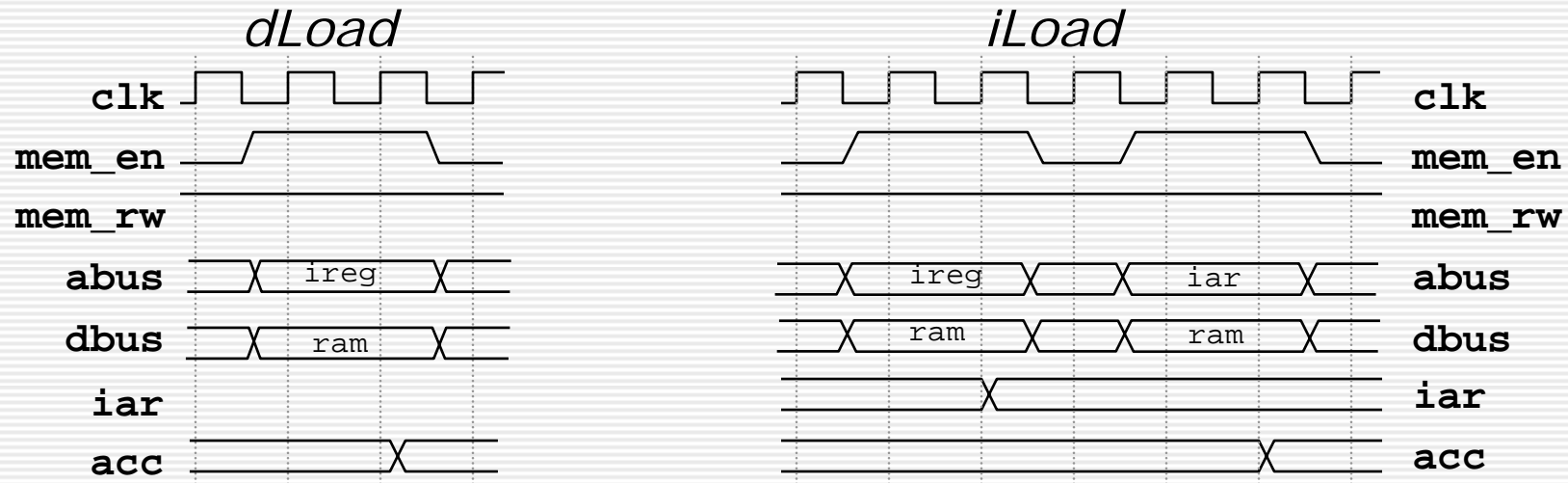
# Add Instruction Execution



# Signal Timing for Processor



# Signal Timing for Processor



# Processor VHDL Specification

```
entity cpu is port (  
    clk, reset    : in  std_logic;  
    m_en, m_rw    : out std_logic;  
    aBus          : out std_logic_vector(adrLength-1 downto 0);  
    dBus          : inout std_logic_vector(wordSize-1 downto 0));  
end cpu;
```

```
architecture cpuArch of cpu is
```

```
type state_type is (reset_state,fetch,halt,negate,mload,dload, ...);  
signal state: state_type;  
type tick_type is (t0, t1, t2, t3, t4, t5, t6, t7);  
signal tick: tick_type;
```

```
signal pc: std_logic_vector(adrLength-1 downto 0); -- program counter  
signal iReg:std_logic_vector(wordSize-1 downto 0); -- instr. register  
signal iar:  std_logic_vector(adrLength-1 downto 0); -- ind. addr. reg.  
signal acc:  std_logic_vector(wordSize-1 downto 0); -- accumulator  
signal alu:  std_logic_vector(wordSize-1 downto 0); -- alu output
```

processor state  
and timing  
definitions

primary  
processor  
registers

```
begin
```

```
  alu <= (not acc) + x"0001" when state = negate else  
        acc + dbus when state = add else (alu'range => '0');
```

ALU  
operations

```
process(clk) -- perform actions that occur on rising clock edges
```

```
function nextTick(tick: tick_type) return tick_type is begin
```

```
  case tick is
```

```
    when t0 => return t1; when t1 => return t2; when t2 => return t3;
```

```
    when t3 => return t4; . . . when others => return t0;
```

```
  end case;
```

```
end function nextTick;
```

```
procedure decode is begin
```

```
  case iReg(15 downto 12) is
```

```
    when x"0" =>
```

```
      if iReg(11 downto 0) = x"000" then state <= halt;
```

```
      elsif iReg(11 downto 0) = x"001" then state <= negate;
```

```
      else state <= halt;
```

```
      end if;
```

```
    when x"1" => state <= mload;
```

```
    . . .
```

```
    when x"a" => state <= add;
```

```
    when others => state <= halt;
```

```
  end case;
```

```
end procedure decode;
```

decode  
instructions

```

procedure wrapup is begin
  state <= fetch; tick <= t0;
end procedure wrapup;
begin
  if clk'event and clk = '1' then
    if reset = '1' then
      state <= reset_state; tick <= t0;
      pc <= (pc'range => '0'); iReg <= (iReg'range => '0');
      acc <= (acc'range => '0'); iar <= (iar'range => '0');
    else
      tick <= nextTick(tick) ; -- advance time by default
      case state is
      when reset_state => state <= fetch; tick <= t0;
      when fetch =>
        if tick = t1 then iReg <= dBus; end if;
        if tick = t2 then decode; pc <= pc+'1'; tick <=t0; end if;
      when halt => tick <= t0; -- do nothing
      when negate => acc <= alu; wrapup;
      when mload =>
        if iReg(11) = '0' then acc <= x"0" & ireg(11 downto 0);
        else acc <= x"f" & ireg(11 downto 0); end if;
      wrapup;
    end case;
  end if;
end begin;

```

last step  
of every  
instruction

load acc  
and fetch  
next

sign extend for  
immediate load of  
negative values

```

when dload =>
    if tick = t1 then acc <= dBus; end if;
    if tick = t2 then wrapup; end if;
when iload =>
    if tick = t1 then iar <= dBus; end if;
    if tick = t4 then acc <= dBus; end if;
    if tick = t5 then wrapup; end if;
when dstore =>
    if tick = t4 then wrapup; end if;
when istore =>
    if tick = t1 then iar <= dBus; end if;
    if tick = t7 then wrapup; end if;
when branch =>
    pc <= x"0" & iReg(11 downto 0); wrapup;
when brZero =>
    if acc=x"0000" then pc <=x"0" & iReg(11 downto 0); end if;
    wrapup;
when brPos =>
    if acc(15) = '0' and acc /= x"0000" then
        pc <= x"0" & iReg(11 downto 0);
    end if;
    wrapup;
. . .

```

load *IAR*  
from  
memory

load *ACC*  
from address  
specified by  
*IAR*

```

process(clk) begin -- perform actions for falling clock edges
  if clk'event and clk = '0' then
    if reset = '1' then
      m_en <= '0'; m_rw <= '1';
      aBus <= (aBus'range => '0'); dBus <= (dBus'range => 'Z');
    else
      case state is
      when fetch =>
        if tick = t0 then m_en <= '1'; aBus <= pc; end if;
        if tick = t2 then m_en <= '0'; aBus <= (aBus'range => '0');
        end if;
      when dload =>
        if tick = t0 then
          m_en <= '1'; aBus <= x"0" & iReg(11 downto 0);
        end if;
        if tick = t2 then
          m_en <= '0'; aBus <= (aBus'range => '0');
        end if;
      when iload =>
        if tick = t0 then
          m_en <= '1'; aBus <= x"0" & iReg(11 downto 0);
        end if;
      end case;
    end if;
  end if;
end process;

```

synchronized to falling clock edge

use PC to supply address

use IREG to supply address

```

if tick = t2 then
    m_en <= '0'; aBus <= (aBus'range => '0');
end if;
if tick = t3 then m_en <= '1'; aBus <= iar; end if;
if tick = t5 then
    m_en <= '0'; aBus <= (abus'range => '0');
end if;
when dstore =>
    if tick = t0 then
        m_en <= '1'; aBus <= x"0" & iReg(11 downto 0);
    end if;
    if tick = t1 then m_rw <= '0'; dBus <= acc; end if;
    if tick = t3 then m_rw <= '1'; end if;
    if tick = t4 then m_en <= '0';
        aBus <= (abus'range => '0'); dBus <= (dBus'range => 'Z');
    end if;
    . . .
end case;
end if;
end if;
end process;

```

drop rw after address is stable

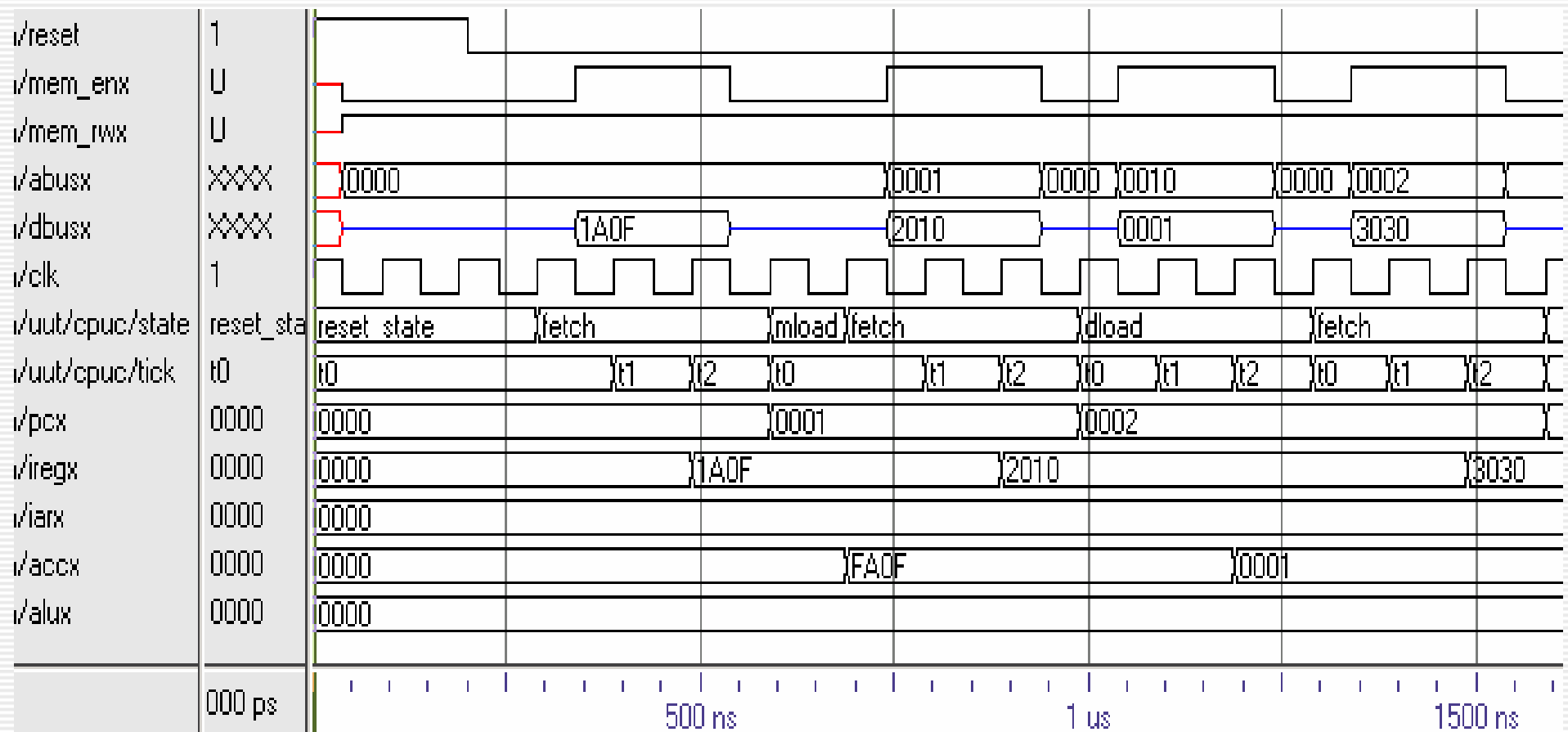
use *IAR* to supply address

raise rw before removing address

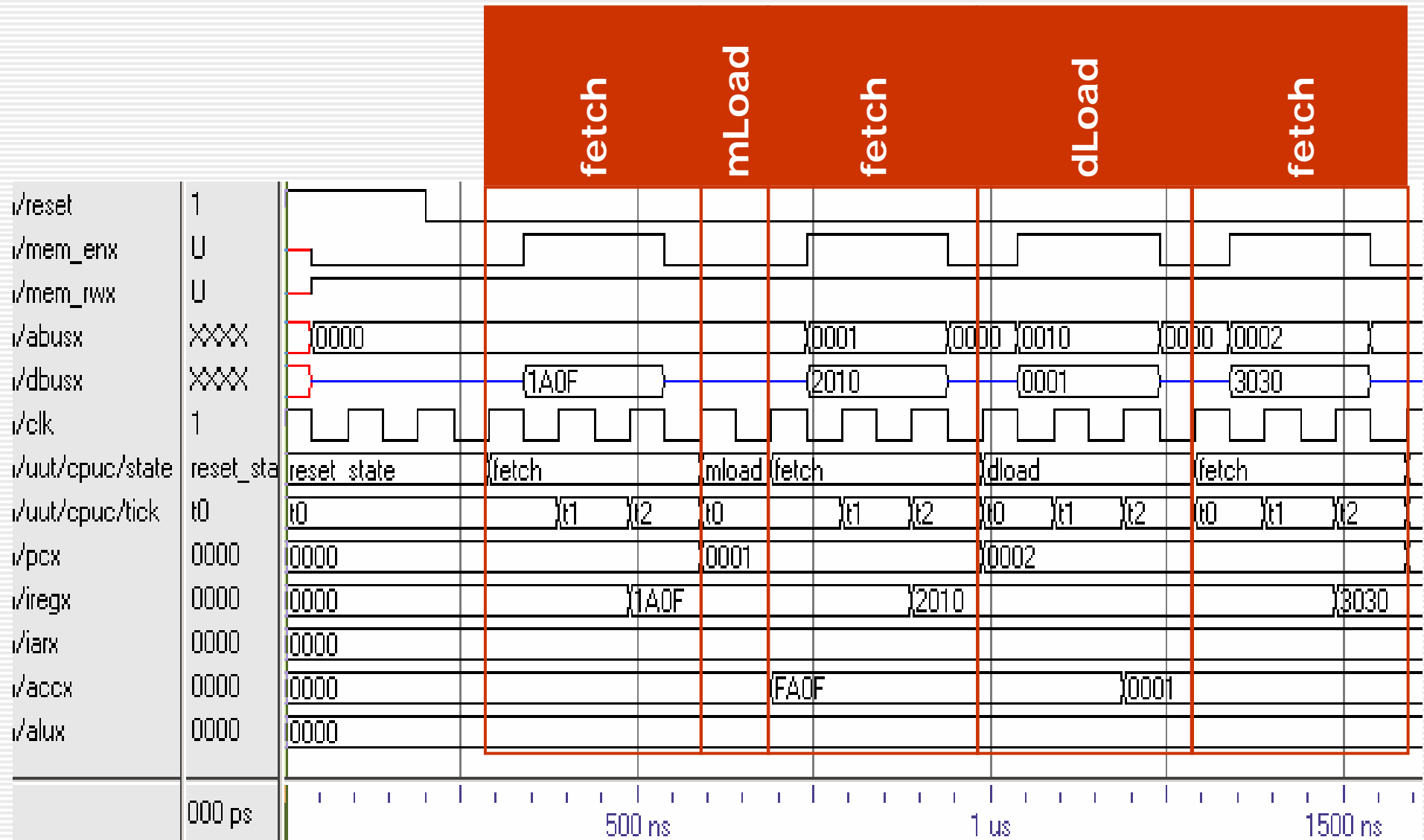
# Processor Test Program

```
ram(0)  <= x"1a0f"; -- immediate load      ram(20) <= x"0000"; -- halt
ram(1)  <= x"2010"; -- direct load         ram(21) <= x"8014"; -- brPos
ram(2)  <= x"3030"; -- indirect load       ram(22) <= x"7014"; -- brZero
ram(3)  <= x"4034"; -- direct store        ram(23) <= x"0001"; -- negate
ram(4)  <= x"0001"; -- negate              ram(24) <= x"9014"; -- brNeg
ram(5)  <= x"2034"; -- direct load         ram(25) <= x"0000"; -- halt
ram(6)  <= x"0001"; -- negate              ram(48) <= x"0031"; -- pointer for iload
ram(7)  <= x"5032"; -- indirect store      ram(49) <= x"5af0"; -- target of iload
ram(8)  <= x"0001"; -- negate              ram(50) <= x"0033"; -- pointer for istore
ram(9)  <= x"1fff"; -- immediate load      ram(51) <= x"0000"; -- target of istore
ram(10) <= x"a008"; -- add                  ram(52) <= x"f5af"; -- target of dstore
ram(11) <= x"700d"; -- brZero
ram(12) <= x"0000"; -- halt
ram(13) <= x"1400"; -- immediate load
ram(14) <= x"8010"; -- brPos
ram(15) <= x"0000"; -- halt
ram(16) <= x"0001"; -- negate
ram(17) <= x"9013"; -- brNeg
ram(18) <= x"0000"; -- halt
ram(19) <= x"6015"; -- branch
```

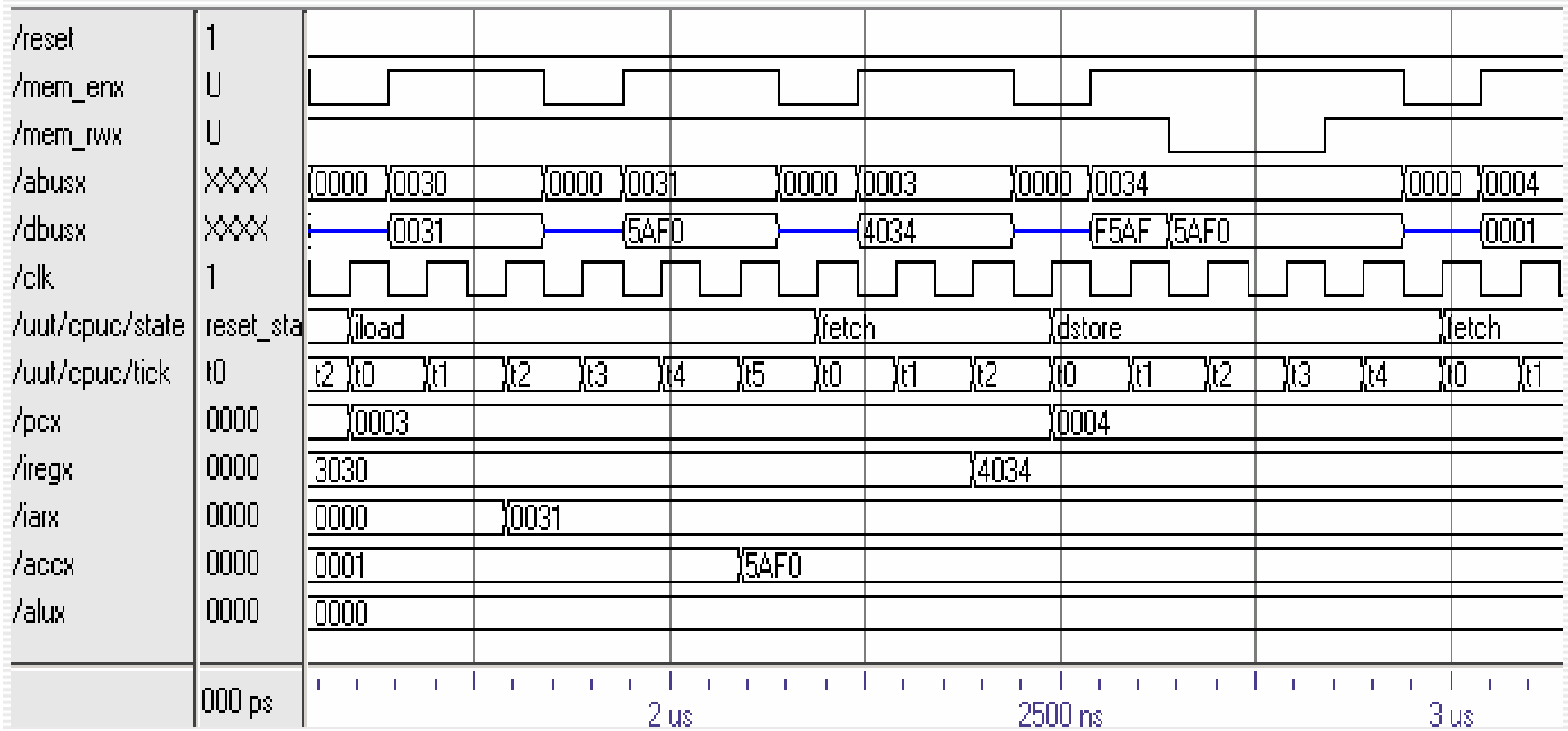
# Processor Simulation



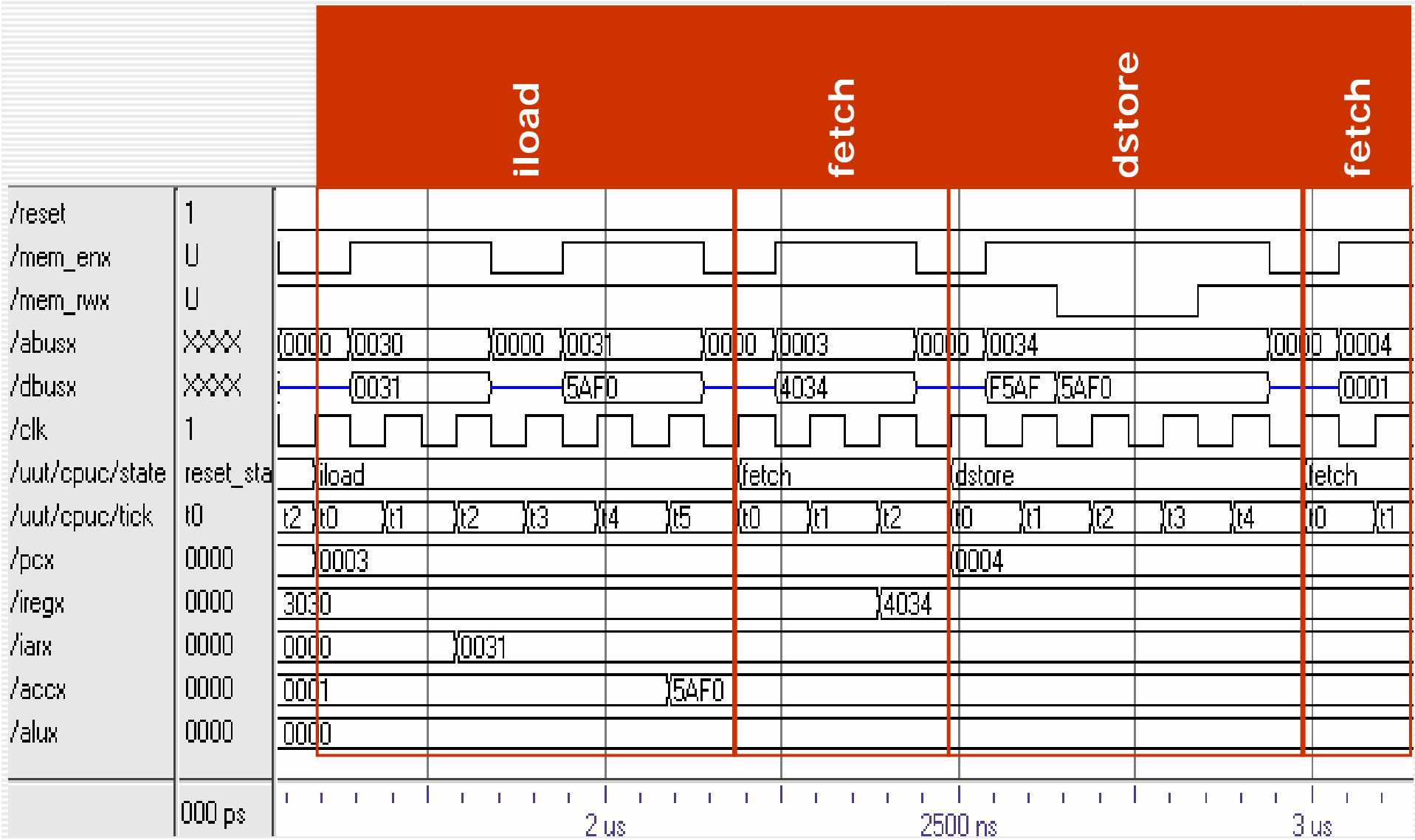
# Processor Simulation



# Processor Simulation



# Processor Simulation



# Processor Timing Simulation

