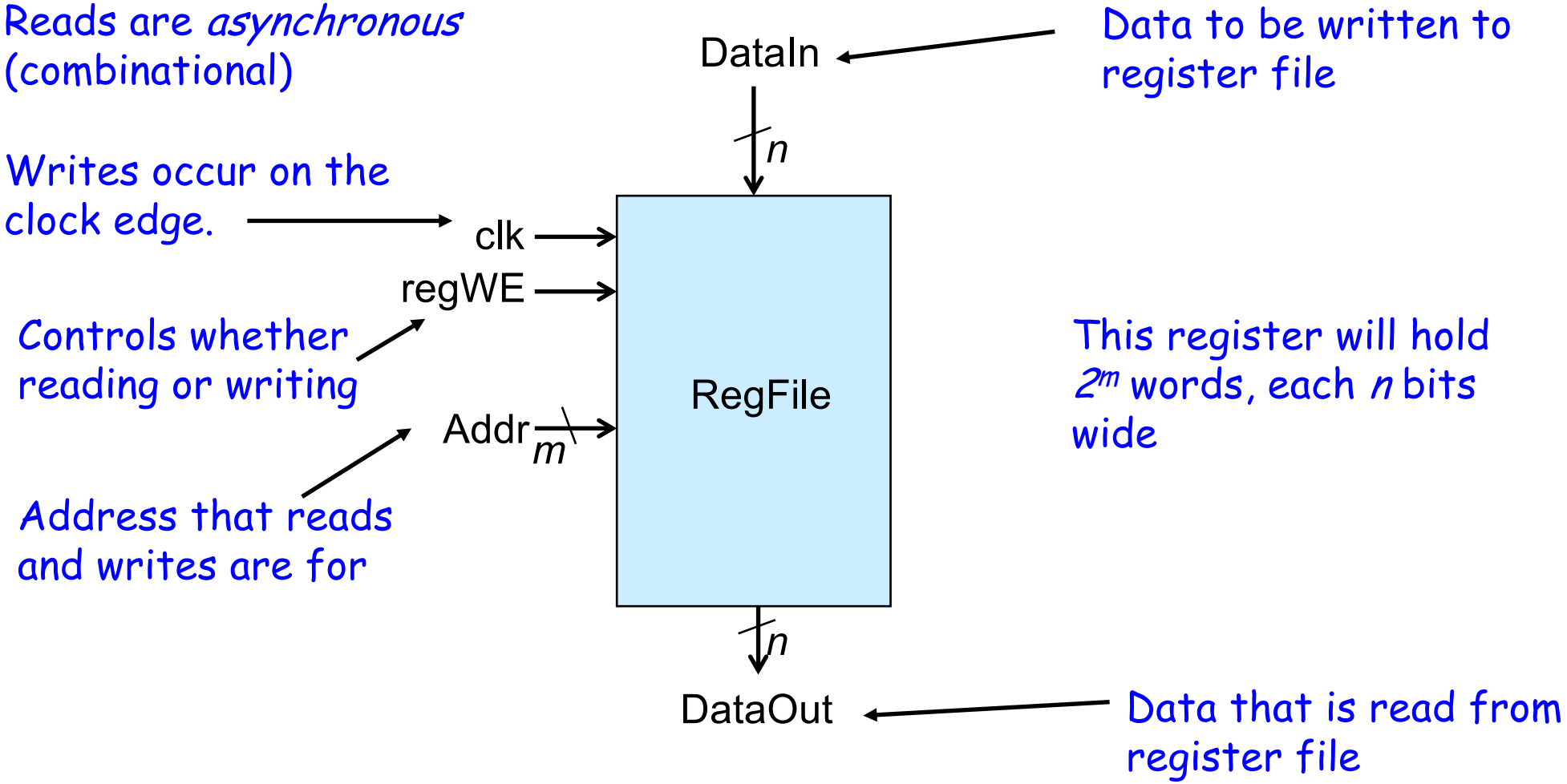
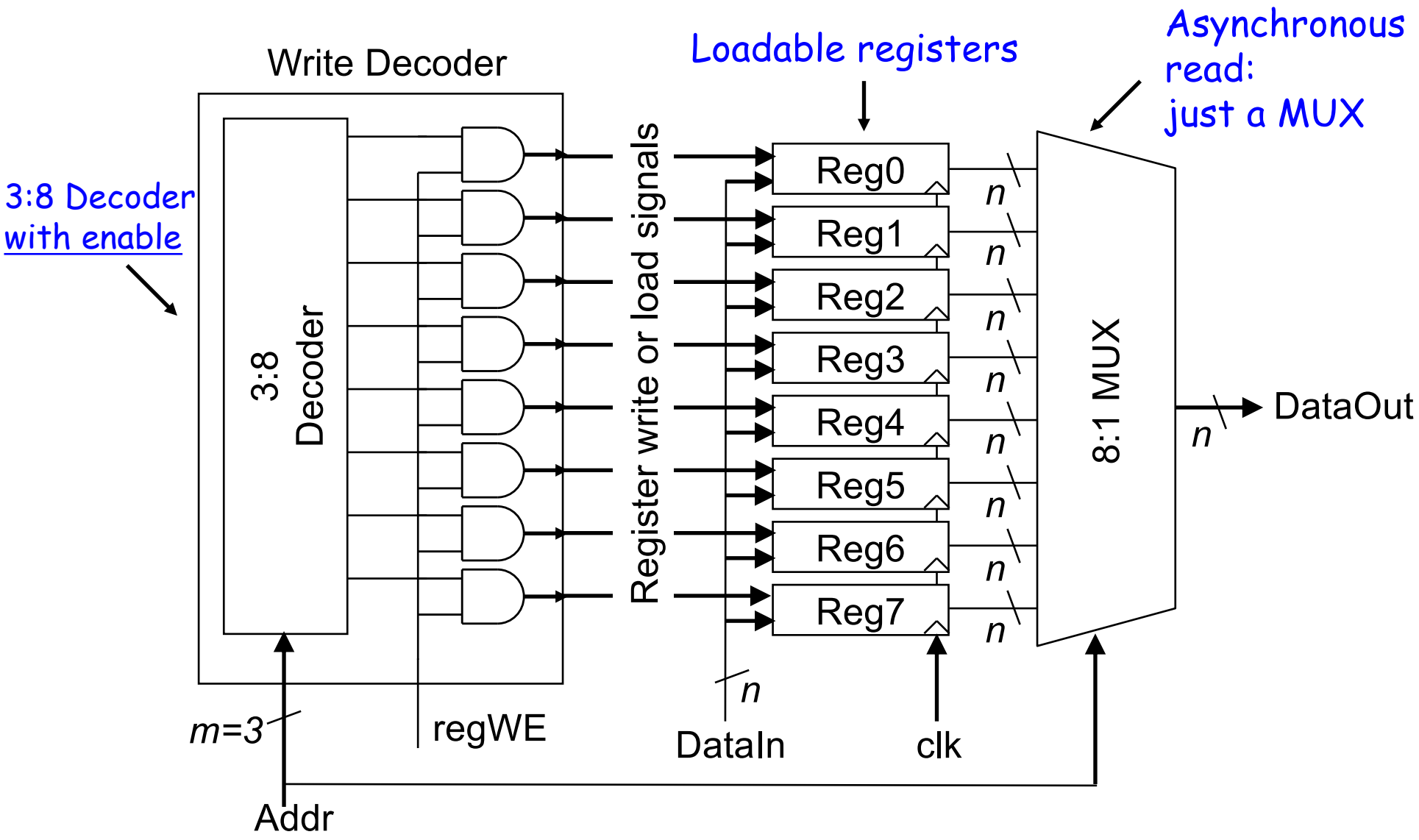
Typical Register File



Building a Register File

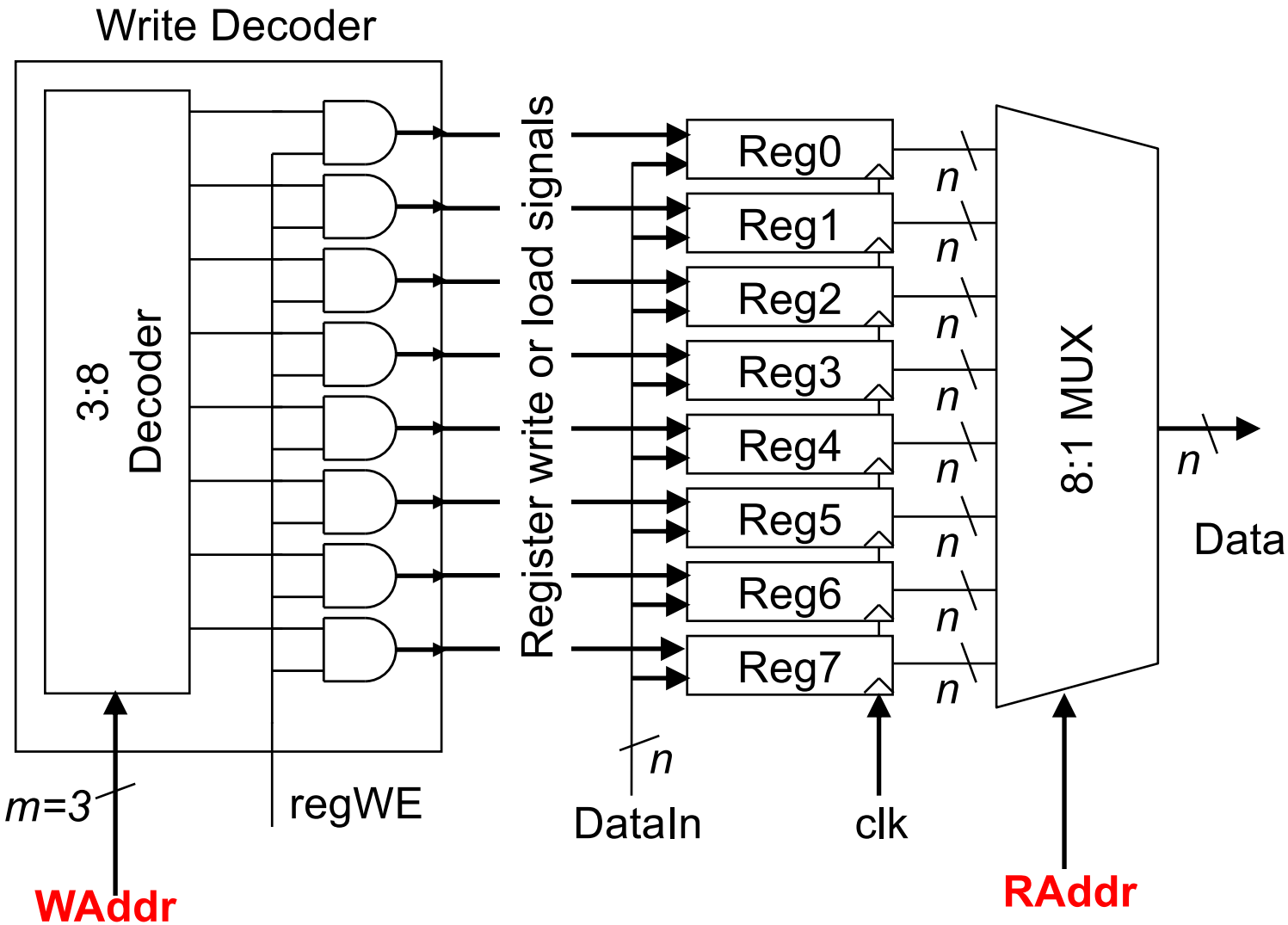


Register File Design Using Behavioral SV

```
module regFile1(
  input logic clk, regWE,
  input logic[2:0] Addr,
  input logic[15:0] DataIn,
  output logic[15:0] DataOut
  );
  logic[15:0] registers[8];

  assign DataOut = register[Addr];

  always_ff @(posedge clk)
  if (regWE)
      registers[Addr] <= DataIn;

endmodule
```

SystemVerilog array of 8. Each location holding 16 bits.

Multi-Ported Register File



One write port, one read port.

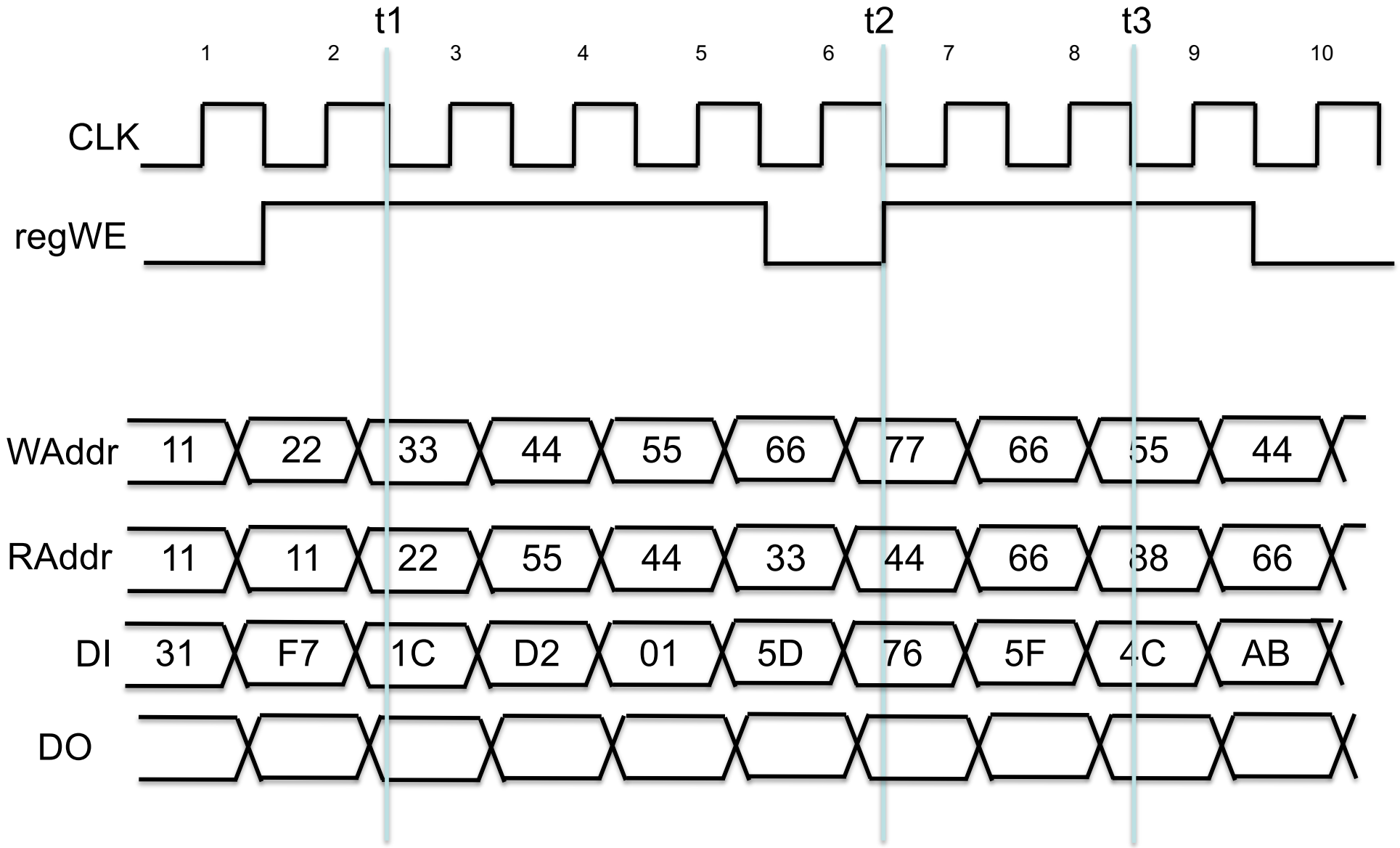
Different write and read addresses

Can be reading from one location on same cycle you write to another location

Register File Design Using Behavioral SV

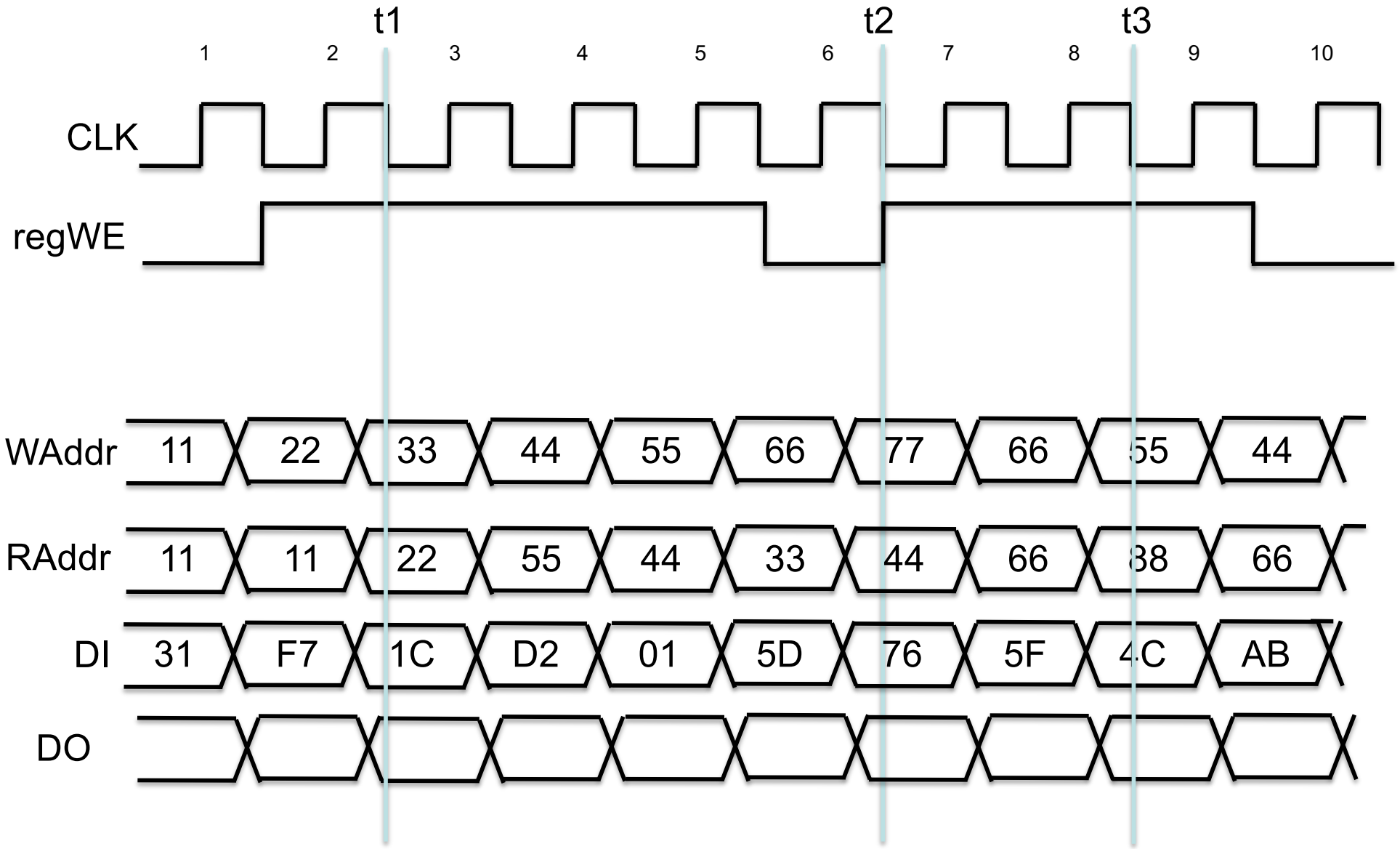
```
module regFile2(
  input logic clk, regWE,
  input logic[2:0] WAddr, Raddr,
  input logic[15:0] DataIn,
  output logic[15:0] DataOut
  );
  logic[15:0] registers[8];

  assign DataOut = register[RAddr];

  always_ff @(posedge clk)
  if (regWE)
      registers[WAddr] <= DataIn;

endmodule
```

SystemVerilog array of 8. Each location holding 16 bits.



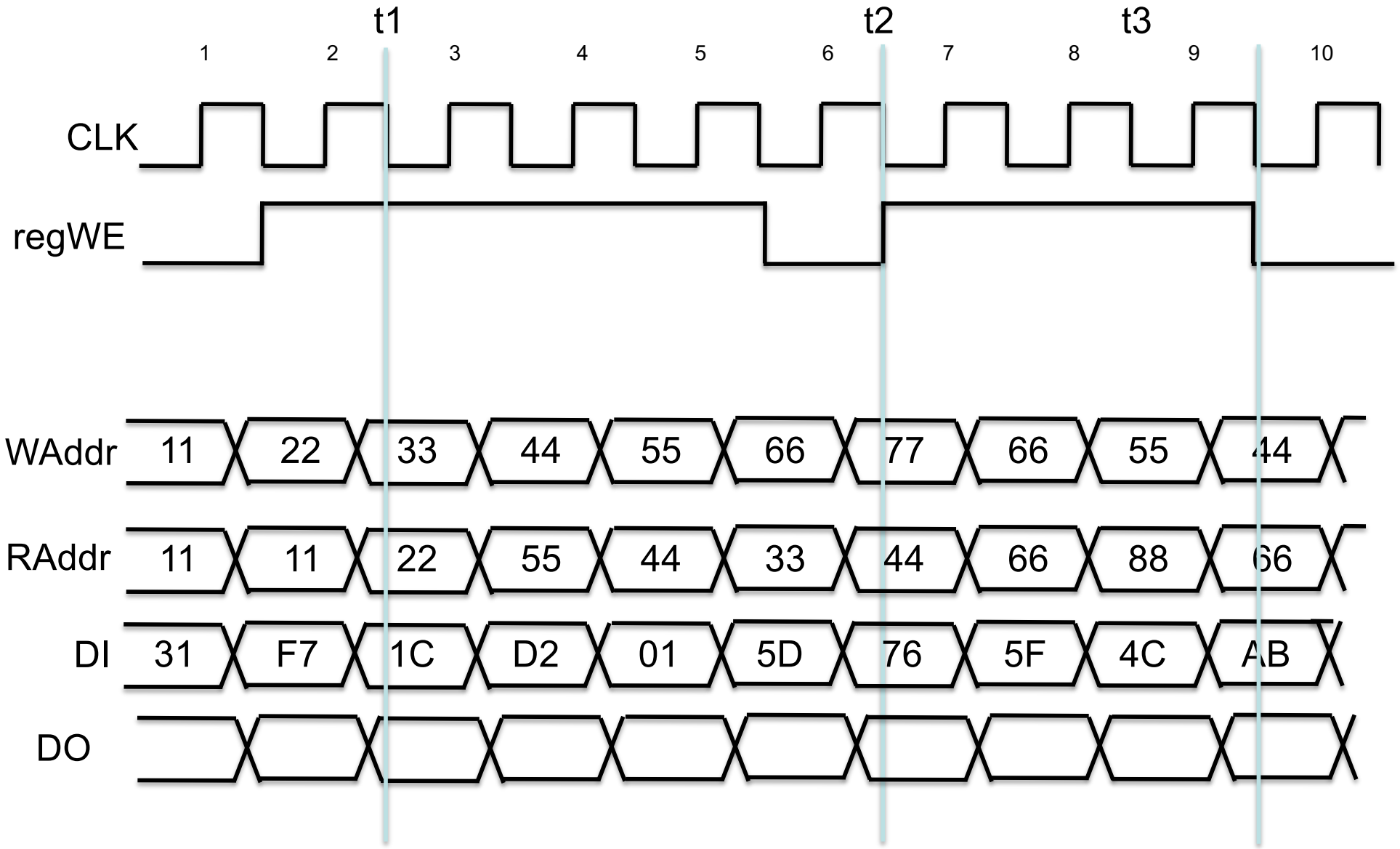
What is the value of DO at time t1?

A) 1C B) 22 C) F7 D) 31
E) unknown



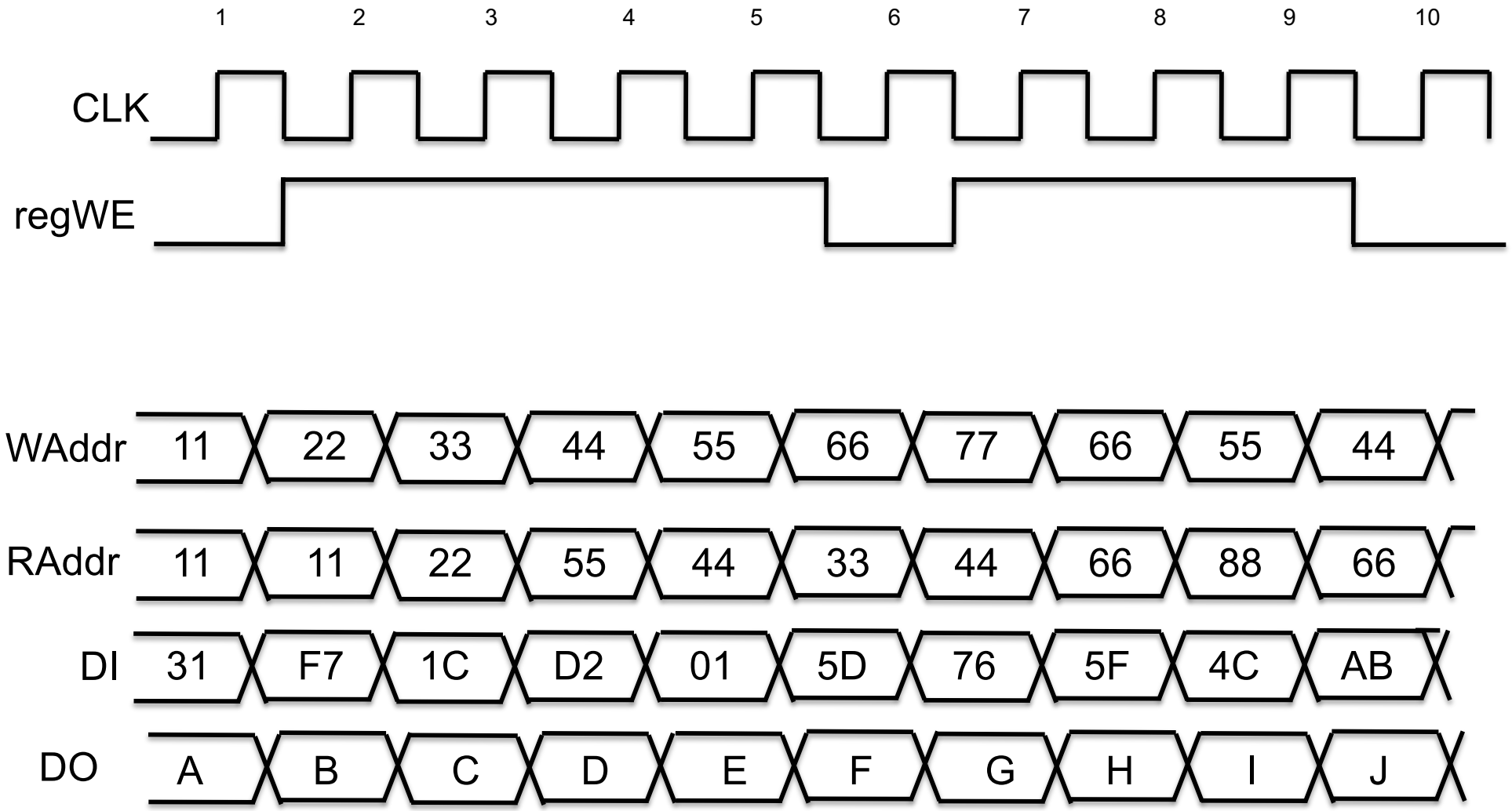
What is the value of DO at time t2?

A) 1C B) 22 C) F7 D) D2
E) unknown

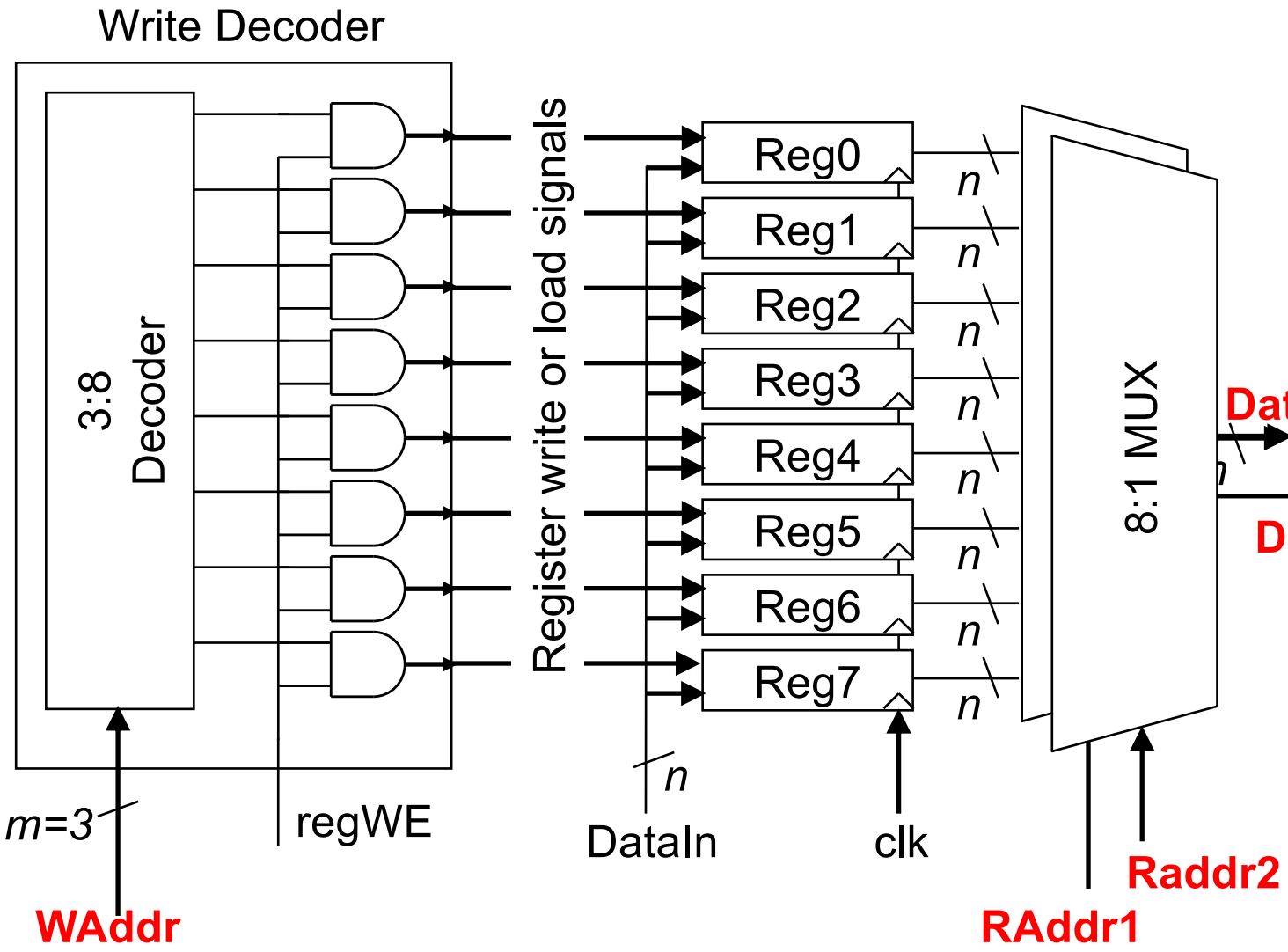


What is the value of DO at time t3?

A) 5F B) 5D C) 76 D) 01
E) unknown


1
2
3
4
5
6
7
8
9
10
CLK
regWE
WAddr
11
22
33
44
55
66
77
66
55
44
RAddr
11
11
22
55
44
33
44
66
88
66
DI
31
F7
1C
D2
01
5D
76
5F
4C
AB
DO
A
B
C
D
E
F
G
H
I
J

Multi-Ported Register File



One write port
Two read ports

Can be reading from two locations on same cycle you write to another location

Useful for microprocessor design

Multi-Ported Register File

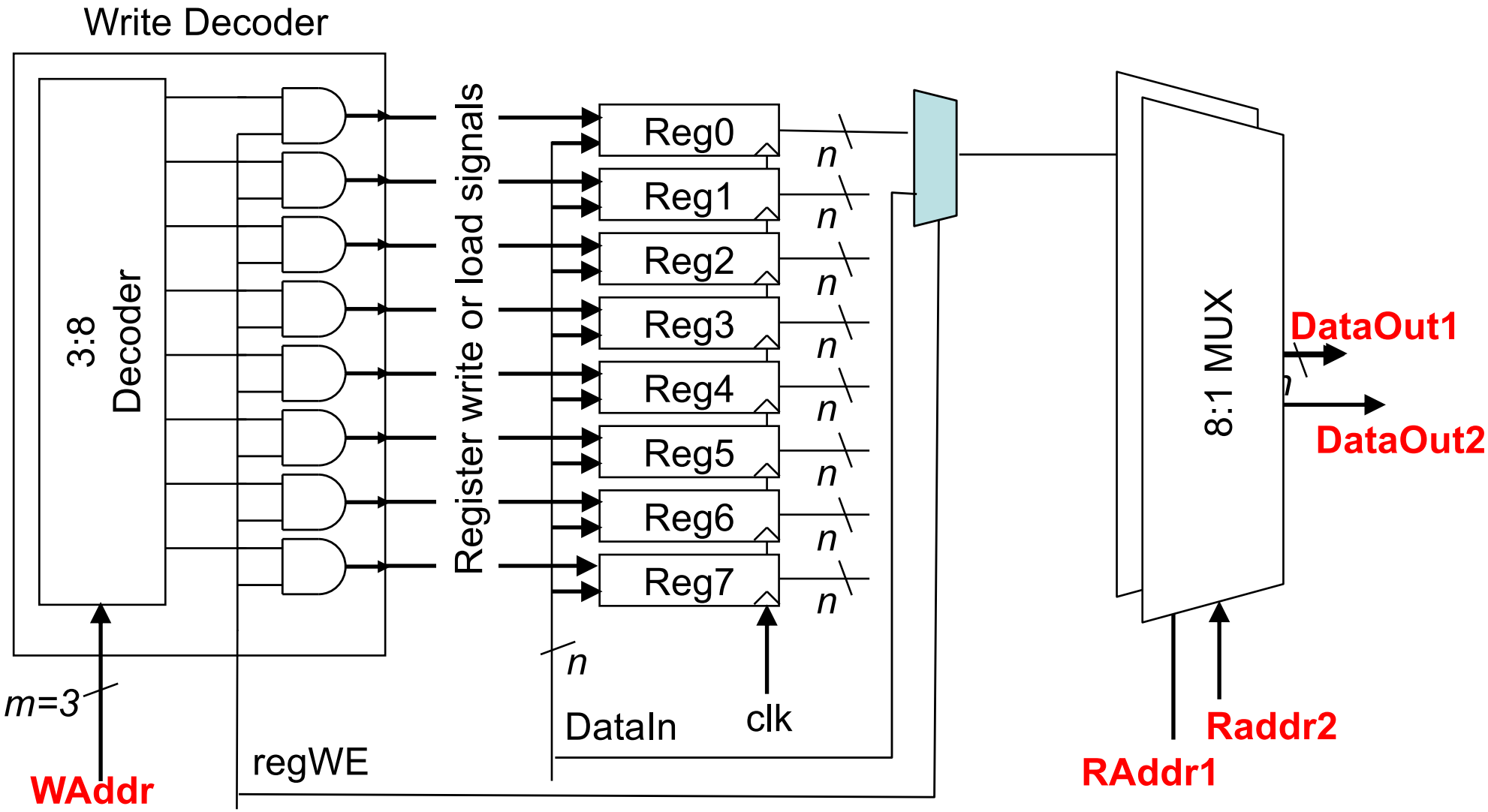
```
module regFile2(
  input logic clk, regWE,
  input logic[2:0] WAddr, Raddr1, RAddr2,
  input logic[15:0] DataIn,
  output logic[15:0] DataOut1, DataOut2
  );
  logic[15:0] registers[8];

  assign DataOut1 = register[RAddr1];
  assign DataOut2 = register[RAddr2];

  always_ff @(posedge clk)
  if (regWE)
      registers[WAddr] <= DataIn;

endmodule
```

SystemVerilog array of 8. Each location holding 16 bits.

Multi-Ported Register File with Bypass

```
module regFile2(
  input logic clk, regWE,
  input logic[2:0] WAddr, Raddr1, RAddr2,
  input logic[15:0] DataIn,
  output logic[15:0] DataOut1, DataOut2
  );
  logic[15:0] registers[8];

  assign DataOut1 = (regWE==1 && Waddr==RAddr1) ?
                                DataIn : register[RAddr1];
  assign DataOut2 = (regWE==1 && Waddr==RAddr2) ?
                                DataIn : register[RAddr2];

  always_ff @(posedge clk)
  if (regWE)
      registers[WAddr] <= DataIn;

endmodule
```

Multi-Ported Register File with Bypass



A Register File vs. RAM

- Random Access Memory (RAM) is conceptually similar to a register file (it stores data in addressed memory)
- A Register File can store data in a small array (bytes) of addressed memory. It is fast but expensive (cost/bit) memory located near the ALU.
- RAM can store data in a very large (Gbytes) array of addressed memory. It is slow but inexpensive.

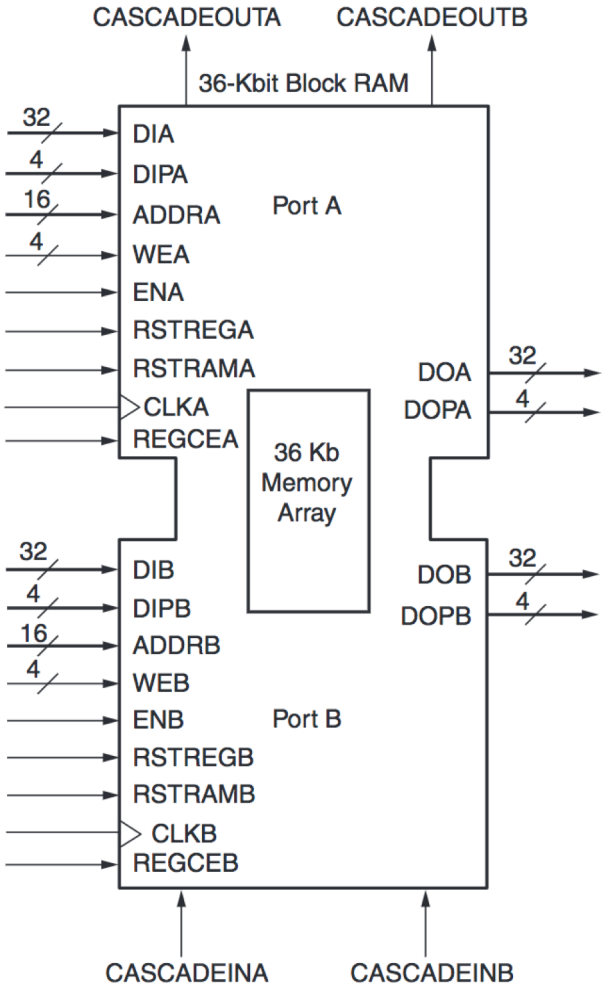
- Larger memories (or even register files) are typically not constructed from flip flops and gates.

- They are hand-crafted transistor-level silicon blocks provided as part of a technology-specific library.

- They may be single-ported or multi-ported. They may be synchronous or asynchronous read or write.

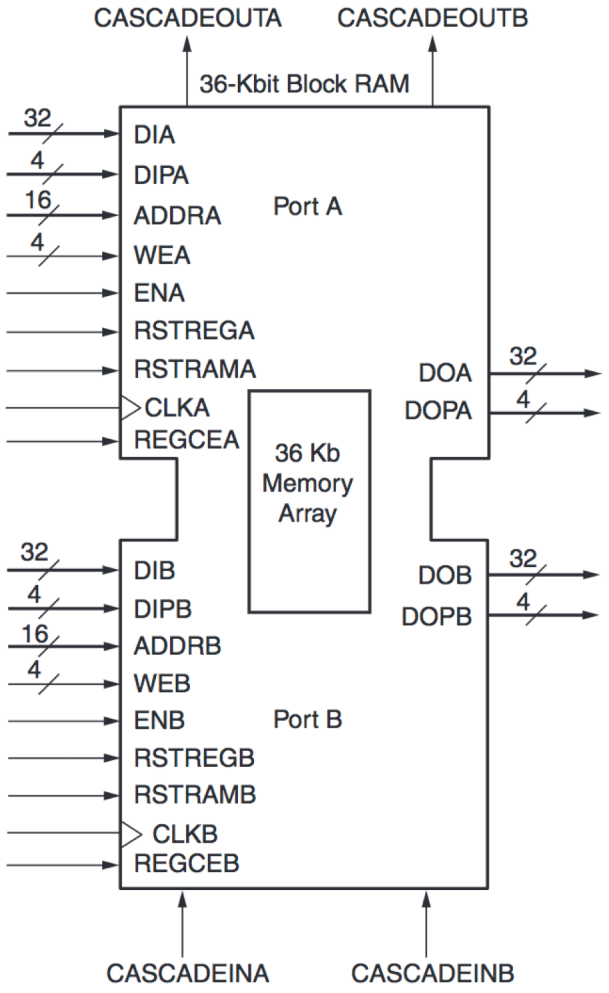
Artix-7 200T Internal Memory

- The FPGA has extensive memory resources
  - Custom "Block Memory"
  - 135 Block RAMs
- Block RAM Features
  - 32 Kb (512 x 64 bits)
    - Additional parity bits available (4 Kb)
  - Dual port (read/write) independent clocks
  - Configurable Size



Block RAM Sizes

- 32Kx1
- 16Kx2
- 8Kx4
- 4Kx8
- 2Kx16
- 1Kx32
- 512x64



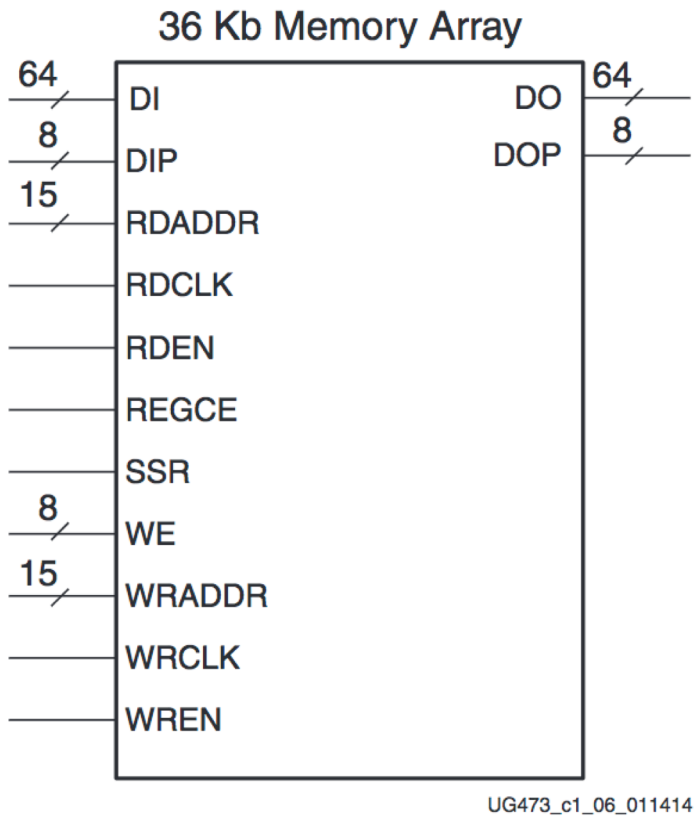
Block RAMSizes

- 32Kx1
- 16Kx2
- 8Kx4
- 4Kx8
- 2Kx16
- 1Kx32
- 512x64

- More address bits needed for narrow memory words
- Some circuit applications need narrow, deep memories
  - Serial, single-bit data sequence
- Some applications need shallow wide memories
  - Interfacing to a 72-bit DRAM Memory

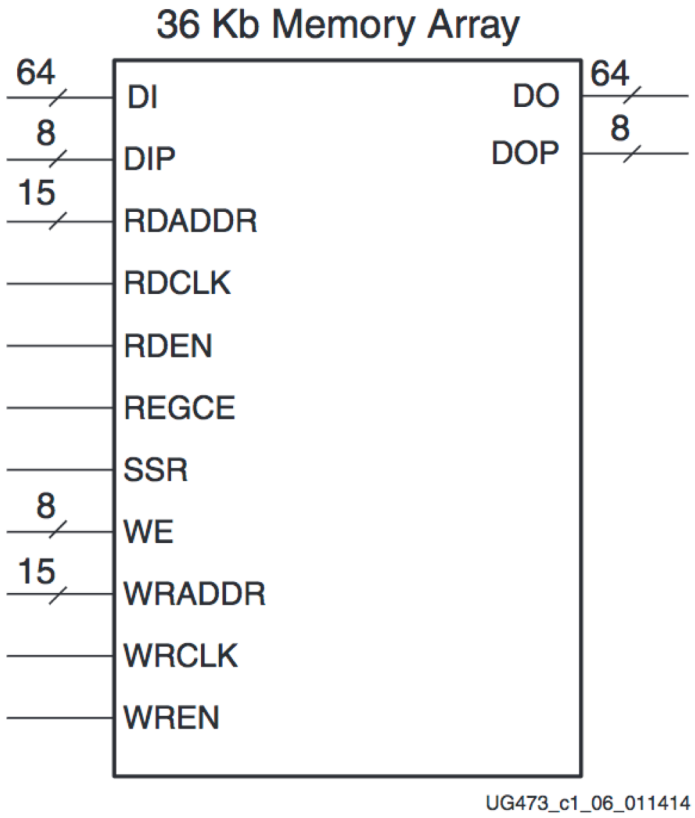
Memory Ports: Reading

- DO: Data Out
  - DOP (Data Out Parity)
- RDCLK: Read Clock
- RADDR: Read Address
- RDEN: Read Enable



Memory Ports: Writing

- DI: Data In (Write Data)
- WE: Write Enable
  - 1 bit for each byte
- WRADDR: Write Address
- WRCLK: Write Clock
  - Writes and reads can have different clocks (most use the same clock)
- WREN: Write Enable
  - Issue a write



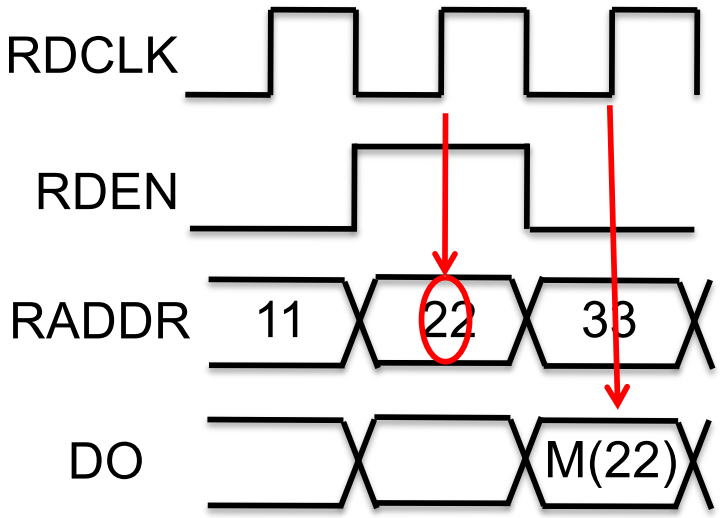
Synchronous RAM

- Small or simple designs could be coded using SystemVerilog but not necessary or recommended.
- Most vendors' tools are able to map the SystemVerilog descriptions to the built-in memories.

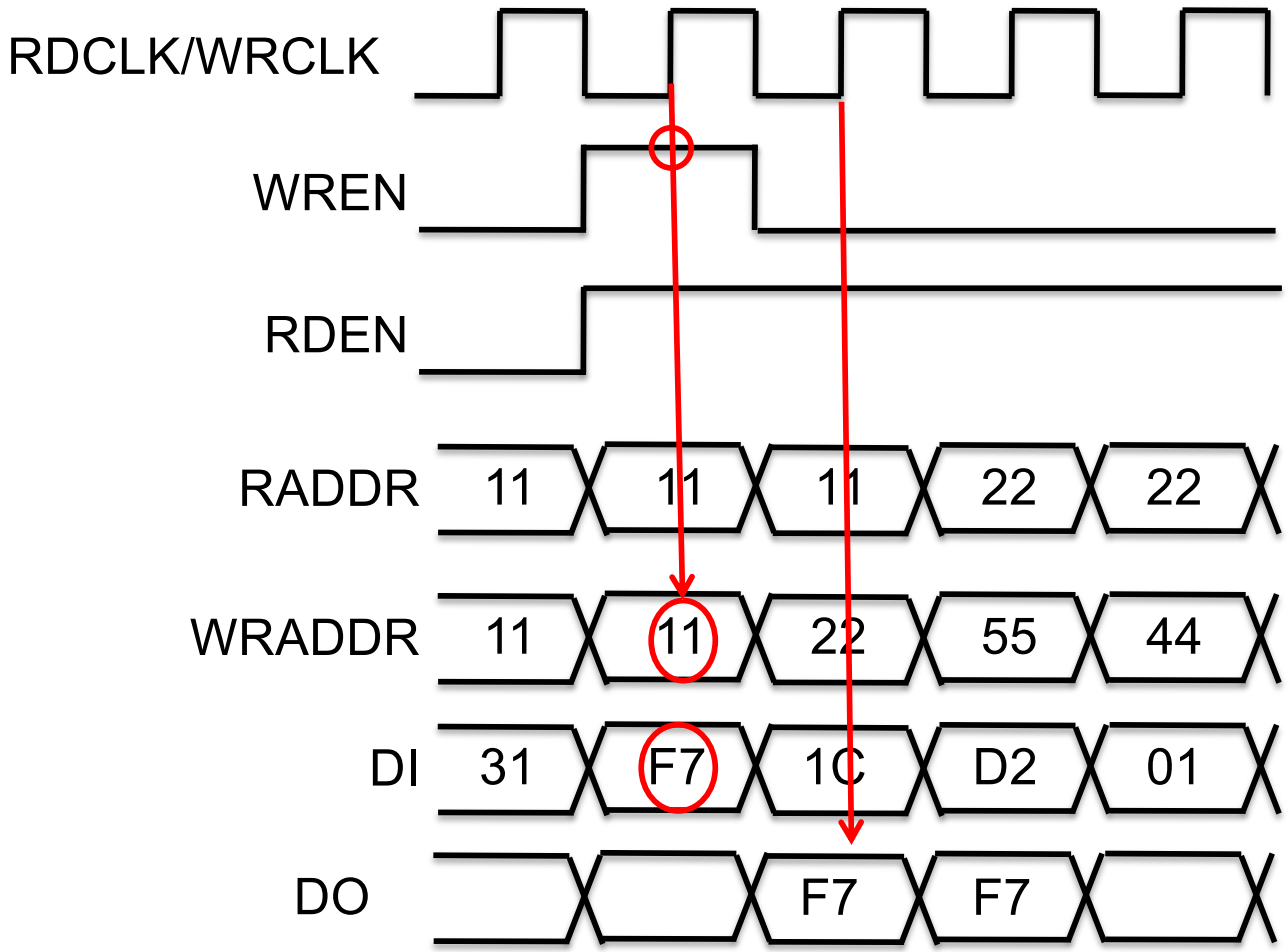
```
module SynchRAM(
  input logic clk, regWE,
  input logic[9:0] Addr,
  input logic[15:0] DataIn,
  output logic[15:0] DataOut
  );
  logic[15:0] registers[1024];

  always_ff @(posedge clk)
  begin
    if (regWE)
      registers[Addr] <= DataIn;
    DataOut <= registers[Addr];
  end
endmodule
```

Read Timing

- Address clocked in on rising clock edge
- Memory output after next rising clock



Write Timing

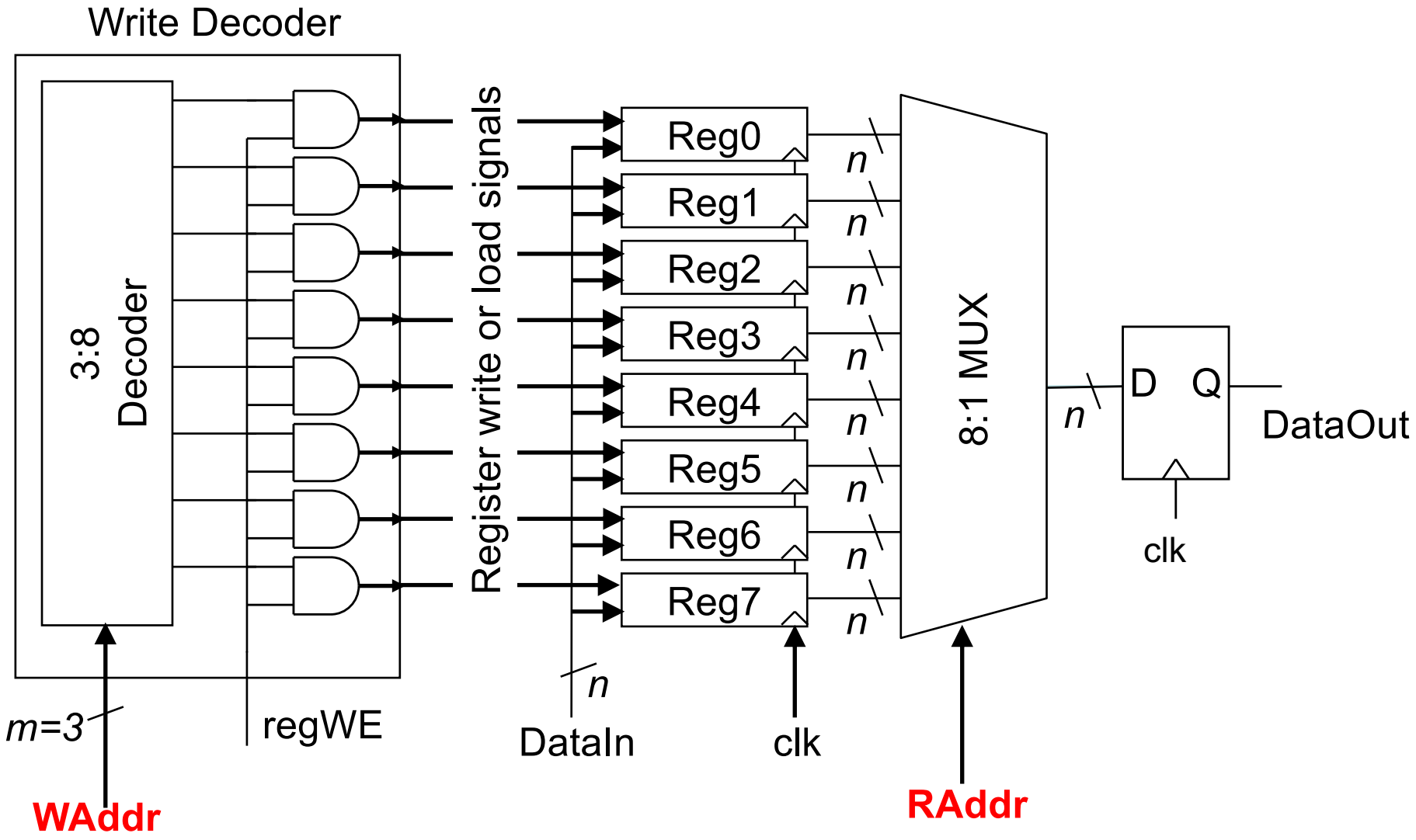


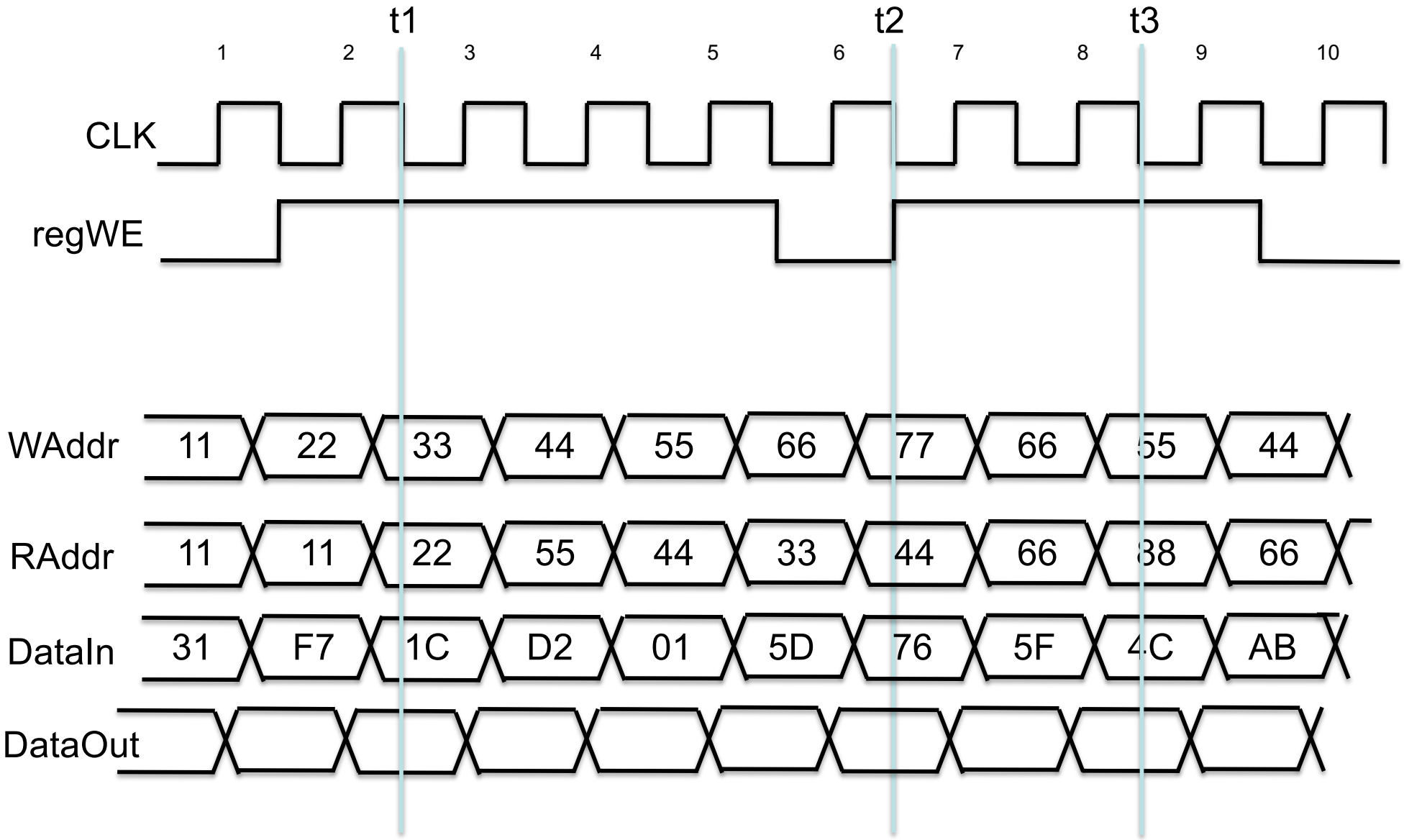
Multi-Ported Synchronous RAM

- Small or simple designs could be coded using SystemVerilog but not necessary or recommended.
- Most vendors' tools are able to map the SystemVerilog descriptions to the built-in memories.

```
module SynchRAM(
  input logic clk, regWE,
  input logic[9:0] WAddr, Raddr,
  input logic[15:0] DataIn,
  output logic[15:0] DataOut
  );
  logic[15:0] registers[1024];

  always_ff @(posedge clk)
  begin
    if (regWE)
      registers[WAddr] <= DataIn;
    DataOut <= registers[RAddr];
  end
endmodule
```

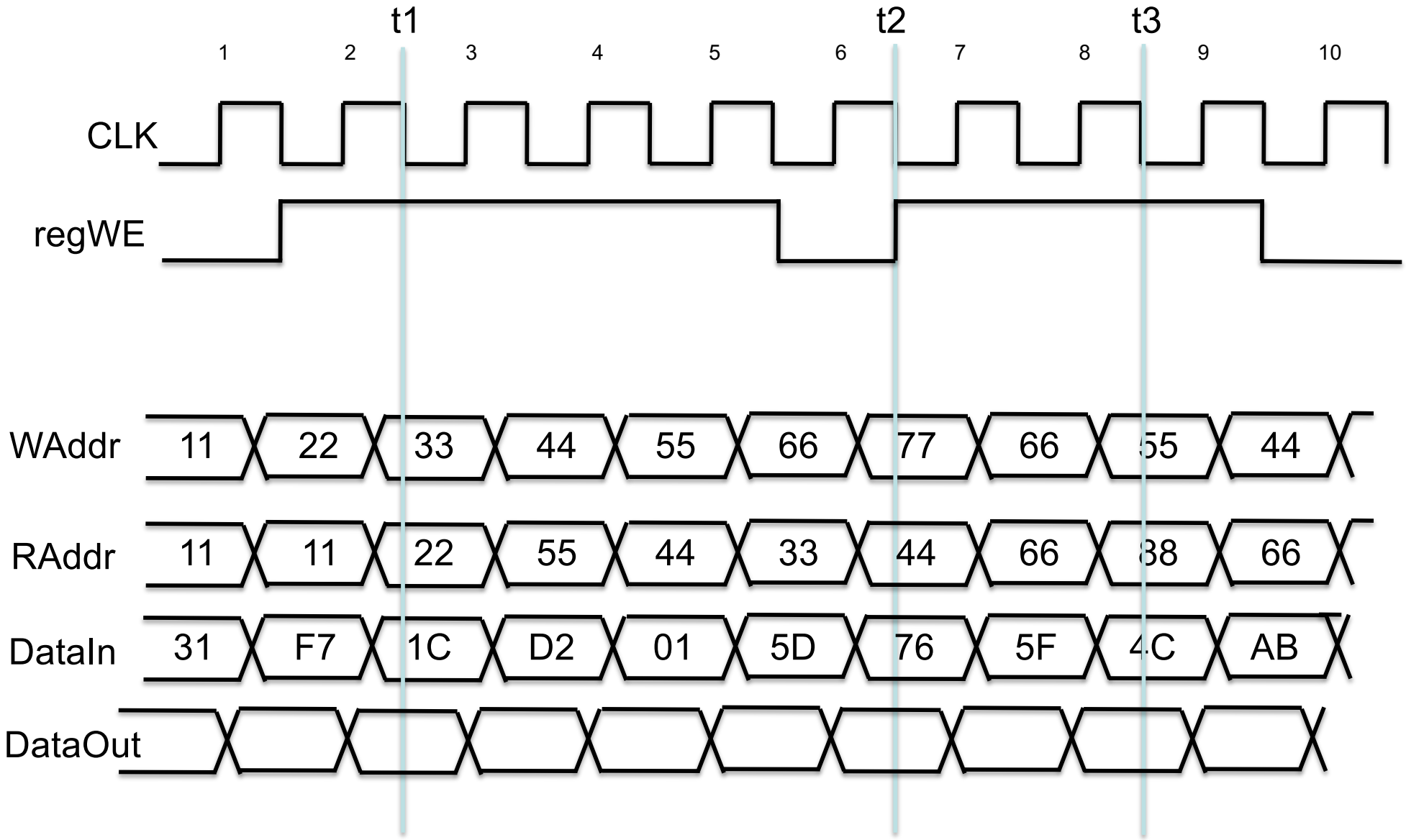
Multi-Ported Synchronous RAM





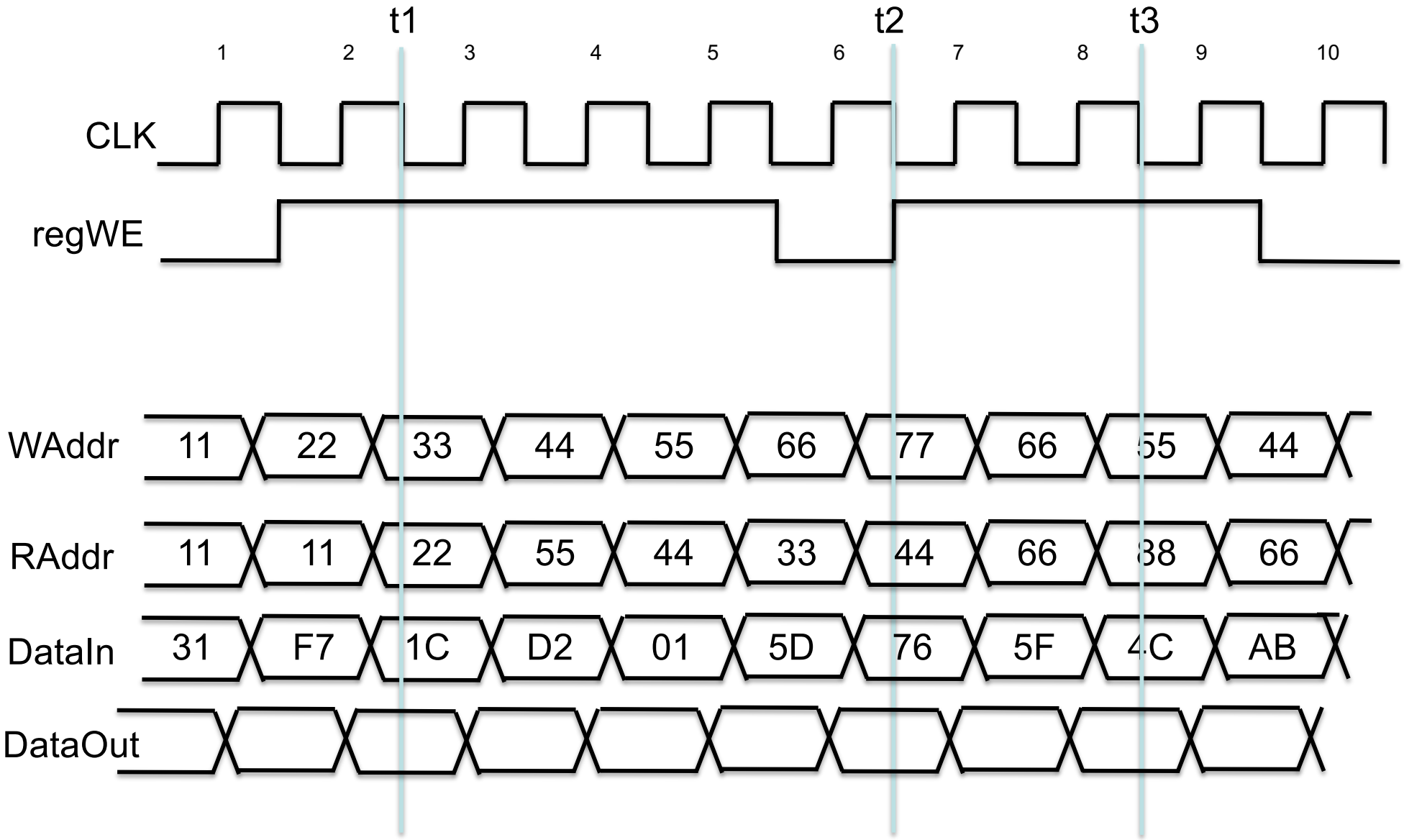
What is the value of DO at time t1?

A) 1C B) 22 C) F7 D) 31
E) unknown



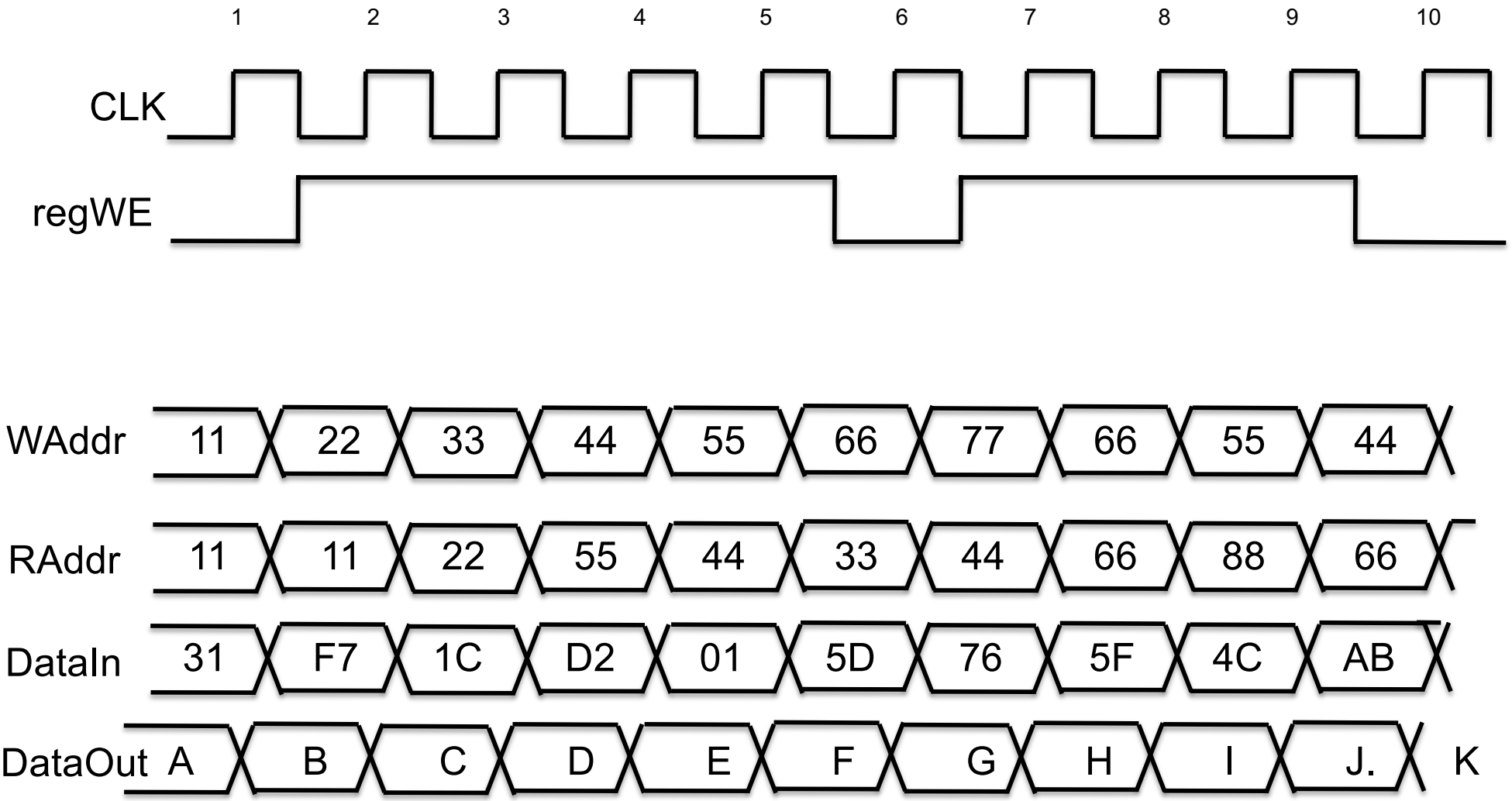
What is the value of DO at time t2?

A) 76 B) 5D C) 1C D) D2
E) unknown



What is the value of DO at time t3?

A) 4C B) 5D C) 1C D) D2
E) unknown


1
2
3
4
5
6
7
8
9
10
CLK
regWE
WAddr
11
22
33
44
55
66
77
66
55
44
RAddr
11
11
22
55
44
33
44
66
88
66
DataIn
31
F7
1C
D2
01
5D
76
5F
4C
AB
DataOut
A
B
C
D
E
F
G
H
I
J.
K

Two Ways to Use BlockRAMs With SystemVerilog

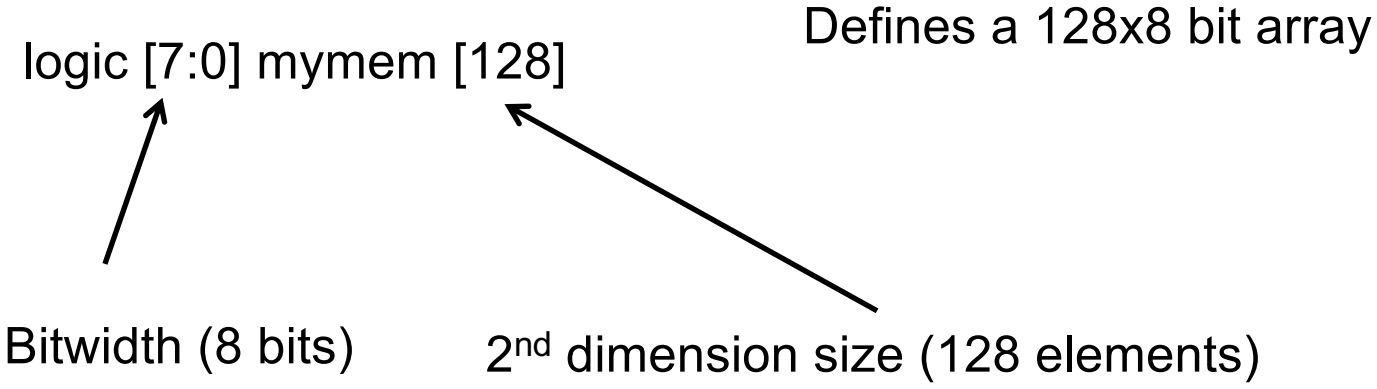
- Instance BlockRAM directly
  - Instance one of the pre-specified BlockRAM library directly within your design
  - *Pros: You get exactly what you want and can customize the memory more than if it is inferred*
  - *Cons: Non-portable. Your code will only work on the specific FPGA you are using, a bit cumbersome (lots of ports, params)*
- *Infer* BlockRAM by proper coding
  - Write your Verilog code so that the synthesis tool recognizes that you want to use a BlockRAM
  - *Pros: Your code is more portable and useable for more FPGAs*
  - *Cons: You may not properly infer the memory you want*

Inferring BlockRAM

- Describe the functionality of a memory using generic SystemVerilog
  - Not specific for a specific memory
  - Easier to read and understand
- Must be written carefully and conform to certain coding styles
  - Synthesis tool will try to map memory descriptions to native memory
  - Will not map to the memory unless functionality is the same

SystemVerilog Two Dimensional Arrays

- SystemVerilog allows the declaration of two dimensional arrays
  - SRAM memory viewed as a two-dimensions
    - Word width (first index)
    - Number of words (second index)
- Example:



Initializing Memory

- You cannot initialize a two dimensional object in SystemVerilog in the declaration:

```
logic [15:0] ram[4] = { 0, 0, 0, 0 };
```

- You can initialize two dimensional objects in an "initial" block
  - "initial" blocks are very similar to "always" blocks but only execute once (for initialization)

```
initial
  for (i=0; i<3; i=i+1)
    ram[i] = 0;
```

Initializing BlockRAM Memory

- Code pattern for a BlockRAM:

```
module myram(
  input logic clk, we,
  input logic[9:0] addr,
  input logic[15:0] di,
  output logic[15:0] dout
  );
  logic[15:0] ram [1024];

  always_ff @(posedge clk)
  begin
    if (we)
      ram[addr] <= di;
    dout <= ram[addr];
  end

endmodule
```

Basic Memory Example

```
module myram(
  input logic clk, we,
  input logic[9:0] addr,
  input logic[15:0] di,
  output logic[15:0] dout
  );
  logic[15:0] ram [1024];

  always_ff @(posedge clk)
  begin
    if (we)
       ram[addr] <= di;
    dout <= ram[addr];
  end

endmodule
```

Read port and write port (shared address)

Declare actual memory cells

Synchronous memory: All memory reads and writes are synchronized to the clock

Reads and writes occur simultaneously

Basic Memory Example

```
module myram(
  input logic clk, we,
  input logic[9:0] addr,
  input logic[15:0] di,
  output logic[15:0] dout
  );
  logic[15:0] ram [1024];

  always_ff @(posedge clk)
  begin
    if (we)
      ram[addr] <= di;
    dout <= ram[addr];
  end

endmodule
```

Do we read the "new" or the "old" value at address 'addr'?

It depends on how it is written

In this example, the "old" value is read (sometime called "read first" mode).

Note that dout gets the "old" value of 'ram[addr]' (ram[addr] is not updated until *after* leaving the always block).

Basic Memory Example #2

```
module myram(
  input logic clk, we,
  input logic[9:0] addr,
  input logic[15:0] di,
  output logic[15:0] dout
  );
  logic[15:0] ram [1024];

  always_ff @(posedge clk)
  begin
    if (we)
      begin
        ram[addr] <= di;
        dout <= di;
      end
    else
      dout <= ram[addr];
  end

endmodule
```

Note that dout gets the "new" value that is being written into ram[addr] when a write occurs.

Basic Memory Example #3

```
module myram(
  input logic clk, we,
  input logic[9:0] addr,
  input logic[15:0] di,
  output logic[15:0] dout
  );
  logic[15:0] ram [1024];

  always_ff @(posedge clk)
    if (we)
        ram[addr] <= di;

  assign dout = ram[addr];

endmodule
```

This code will NOT map to a BlockRAM because it does not support asynchronous mode

The memory read is “asynchronous” and does not use the clock.

```
module myram(
  input logic clk, we,
  input logic[9:0] addr,
  input logic[15:0] di,
  output logic[15:0] dout
  );
  logic[15:0] ram [1024];

  always_ff @(posedge clk)
  begin
    if (en)
      if (we) begin
          ram[addr] <= di;
          dout <= di;
        end
      else
        dout <= ram[addr];
     end

endmodule
```

Add an “Enable” for reading/writing

```
module myram(
  input logic clk, we, ena, enb,
  input logic[9:0] addra, addrb,
  input logic[15:0] di,
  output logic[15:0] dout
  );
  logic[15:0] ram[1024];

  always_ff @(posedge clk)
  begin
    if (ena)
      if (we) begin
          ram[addra] <= di;
  end

 always_ff @(posedge clk)
   if (enb)
      dout <= ram[addrb]

endmodule
```

Dual ports (different address bus for read and write)