MEMORIES

Memory Outline

- Overview
  - Array Memory View 1
  - Array Memory View 2: Types
    - Read/Write
    - ROM
    - NVRWM
  - Array Memory View 3
    - Combinational
    - Technically sequential
    - Sequential
- 6T SRAM cell
- SRAM circuits and layout
- Multi-port SRAM
- DRAM
- ROM circuits and layout
- Array Memory View 4: memories for ASICs and FPGAs
  - On-chip macros
    - ASICs
    - FPGAs
  - On-chip standard cell
    - ASICs
    - FPGAs
  - Off-chip
- Synthesized standard cell memories
  - Verilog
  - Timing
  - ROM: standard cell verilog
  - ROM: FPGA block RAM

Array Memory View 1: A Component of Digital Systems

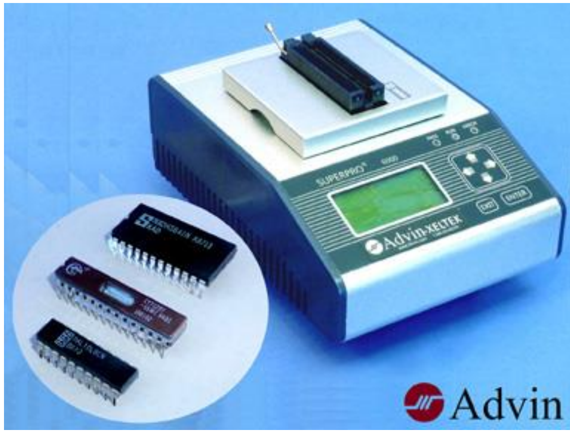
- Three primary components of digital systems
  - Datapath (does the work)
  - Control (manager)
  - **Memory (storage)**
    - "Single bit" ("foreground")
      - Clockless latches e.g., SR latch
      - Clocked transparent latches e.g., D latch
      - Clocked edge-triggered flip flops e.g., D FF
    - **"Array" memories** ("background")

Memories

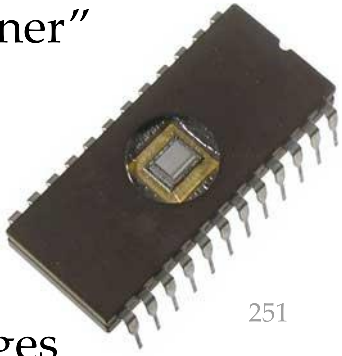
- Use in general digital processors
  - Instructions
  - Data
- Usage is more widespread in DSP, multimedia, embedded processors
  - Buffering input/intermediate/output data (e.g., rate matching)
  - Storing fixed numbers (e.g., coefficients)
  - Often relatively small (e.g., 8-64-256 words) and numerous (dozens are not unusual)
- Key design goal: density, especially for the memory cells. This means fitting the largest amount of memory storage into a certain amount of chip area

Array Memory View 2: Types of Memories

1. Read-write memories
   - SRAM: Static random access memory
     - Data is stored as the state of a bistable circuit typically using "back-to-back" inverters
     - State is retained without refresh as long as power is supplied
   - DRAM: Dynamic random access memory
     - Data is stored as a charge on a capacitor
     - State leaks away, refresh is required



2. ROM: Read-only memory, non-volatile
   - Basic ROM – mask programmed at design time
   - PROM: Programmable read-only memory; typically programmed at manufacture time by a "PROM burner"
     - Using fuse or anti-fuse circuits
   - Synthesized from standard cells



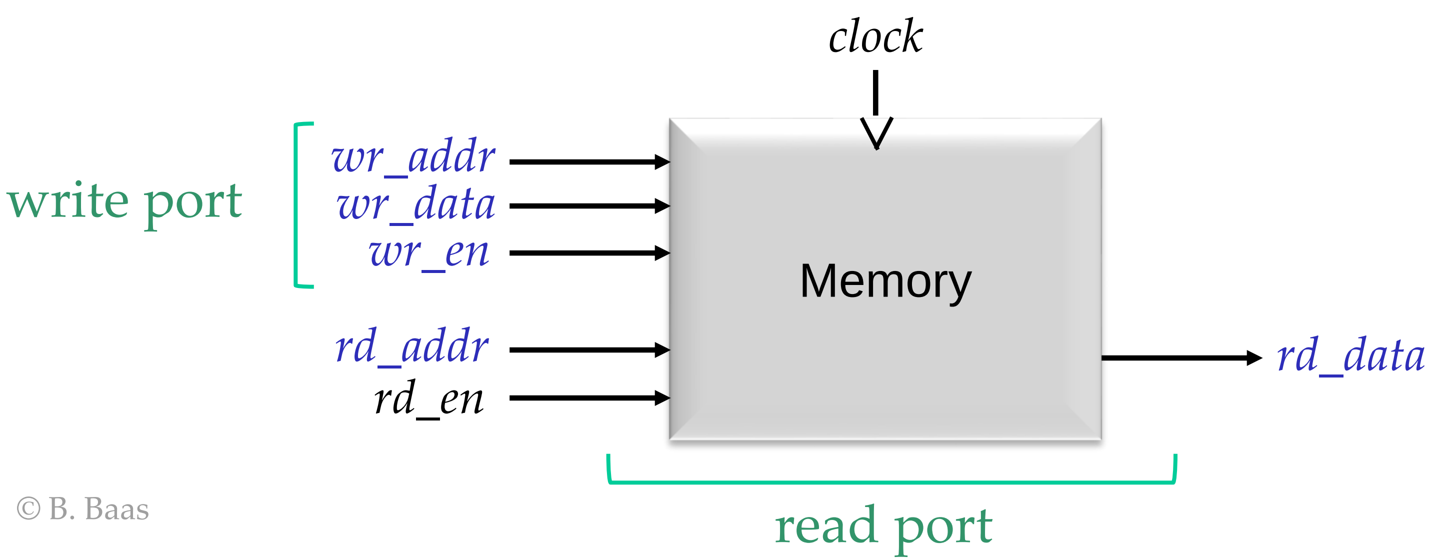
3. NVRWM: Non-volatile read-write memory
   - EPROM: Erasable ROM, erasable with UV light
   - EEPROM/Flash: ROM at low voltages, writable at high voltages

Memory View 3: Memory Logical Categories

- Combinational (output depends on present inputs only)
  - ROM: read-only memory
  - May be straight-through truly-combinational, or registered
- Feels like Combinational but technically Sequential
  - PROM: programmable read-only memory
  - EPROM: ROM, but erasable with UV light
- Sequential (output depends on present and past inputs)
  - SRAM: static memory
  - DRAM: dynamic memory
  - Flash: ROM at low voltages, writable at high voltages

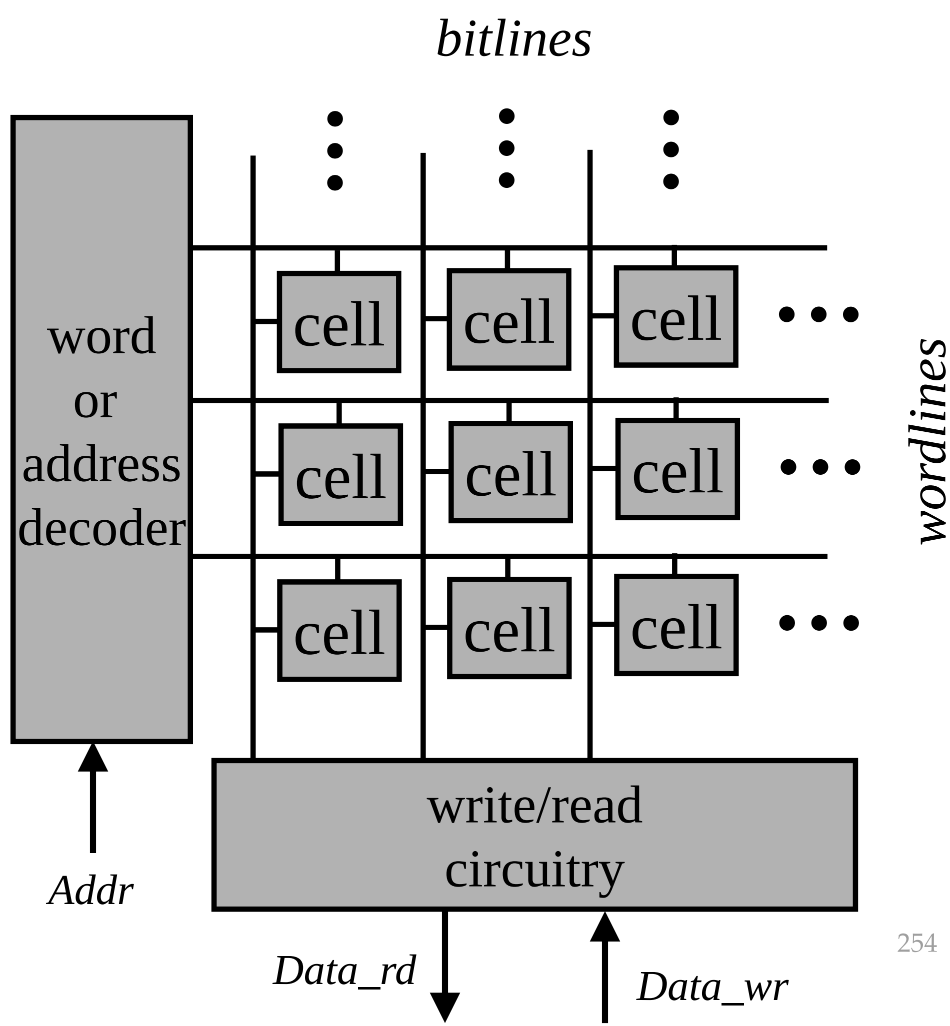
Basic Memory Inputs and Outputs

- The basic memory structure includes a write port and a read port as shown in the figure
  - Clocked or Synchronous memories include a *clock* input
  - A read-enable input (*rd_en*) is not needed for functionality but is often included to enable reduced power dissipation when read operations are not needed



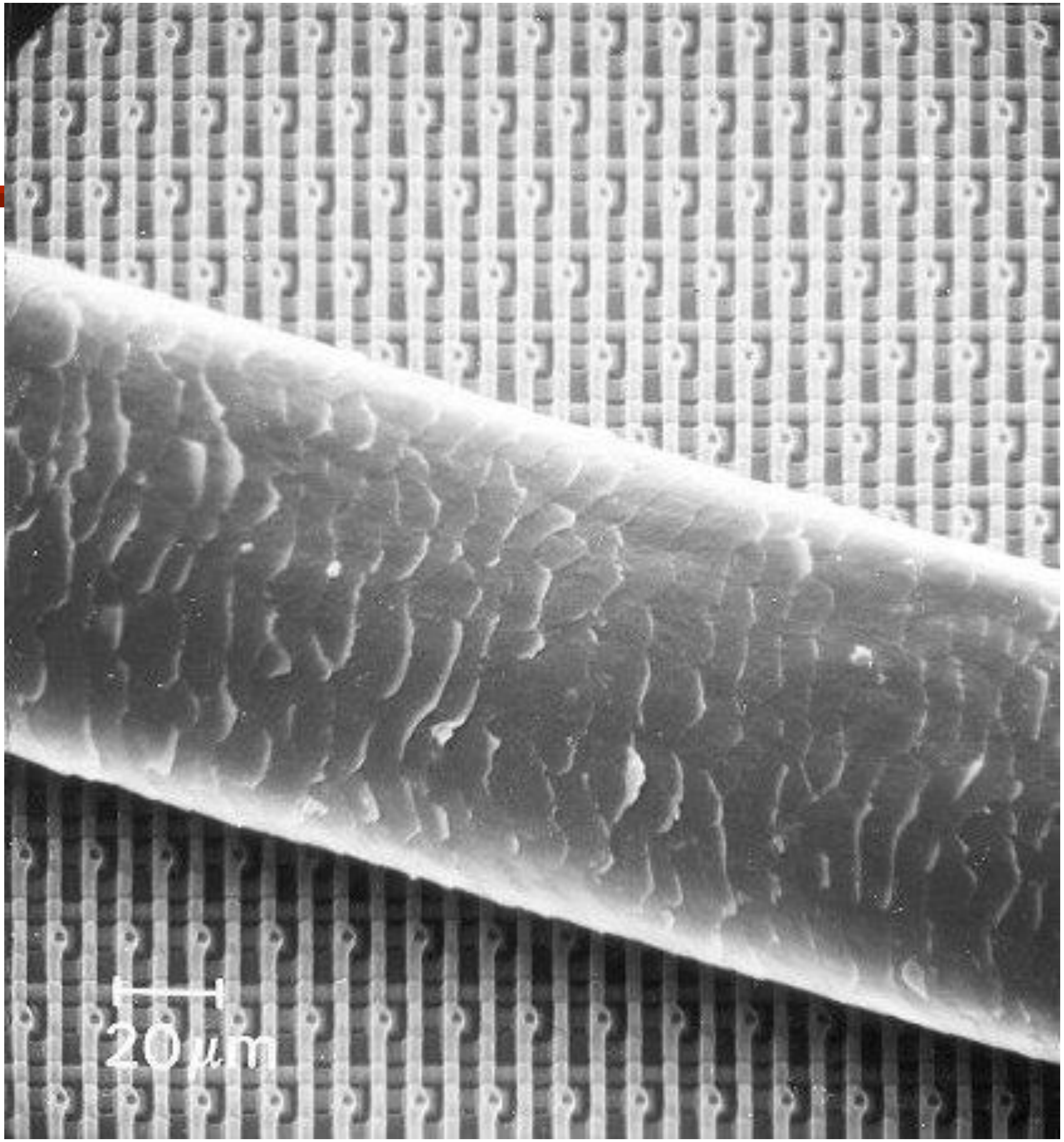
Memories

- Memories generally contain several components:
  - Array of cells
  - Address decoder
  - Write circuitry
  - Read circuitry (sense amplifiers)
  - *wordlines*
  - *bitlines*
- Interface signals
  - *Address* (one for each port)
  - *Data* (one for each port)
  - *Enable_write*
  - *Enable_read* (likely)
  - *Clock* (sometimes)



Memory Array

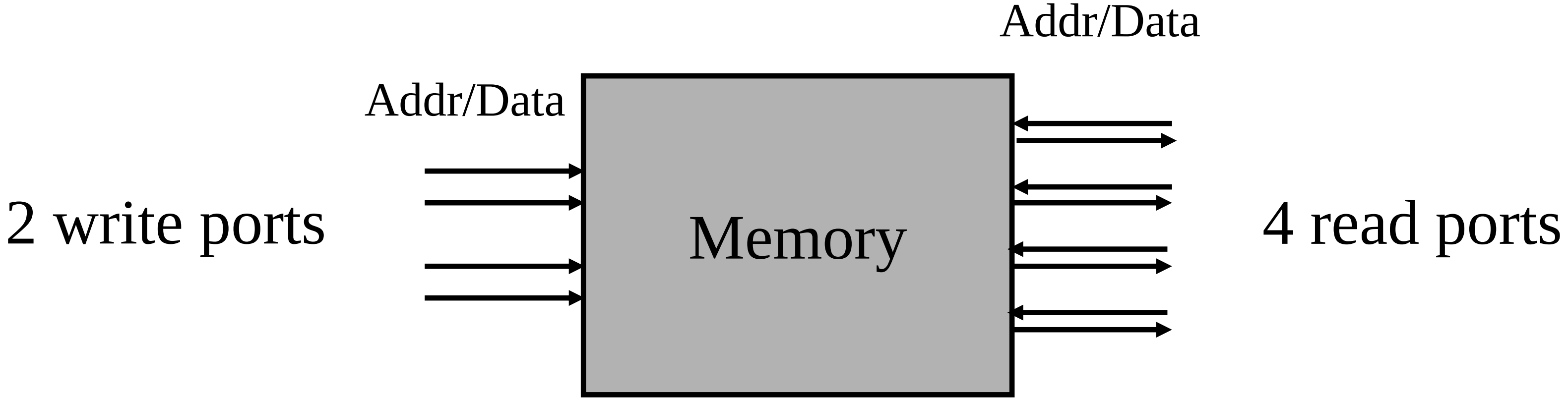
- Human hair on a 256 Kbit memory chip



Source: Helmut Föll

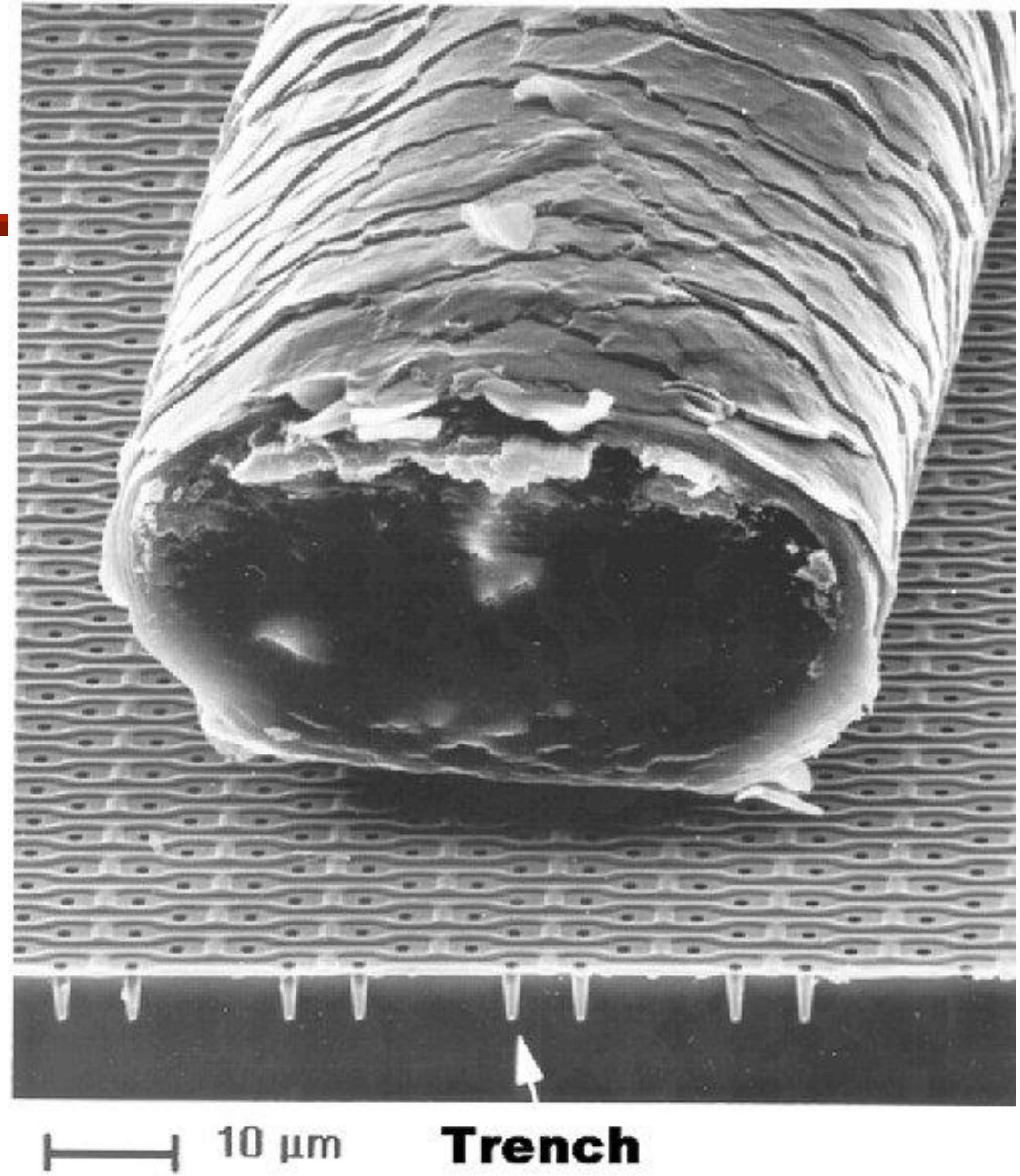
Multi-ported SRAM

- Frequently used in register files
  - Classic RISC computers have 1 write and 2 read ports
  - Modern multiple-instruction-issue computers can have many ports (22 (12 Rd, 10 Wr) in Itanium [ISSCC 05])
- More commonly use single-ended (non-differential) bitlines



Memory Array

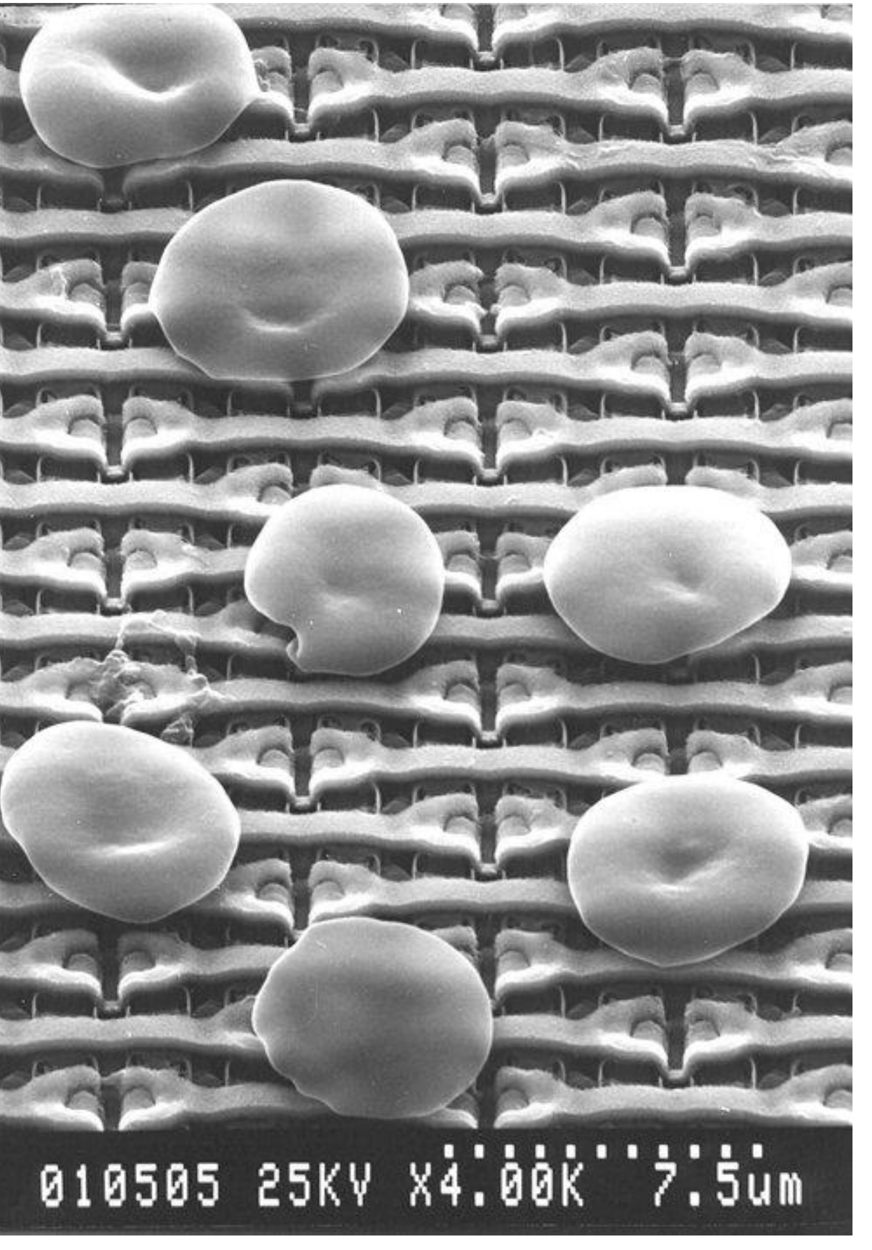
- Human hair on a 4 Mbit memory chip
- Note DRAM trench capacitors



Source: Helmut Föll

Memory Array

- Red blood cells on a 1 Mbit memory chip



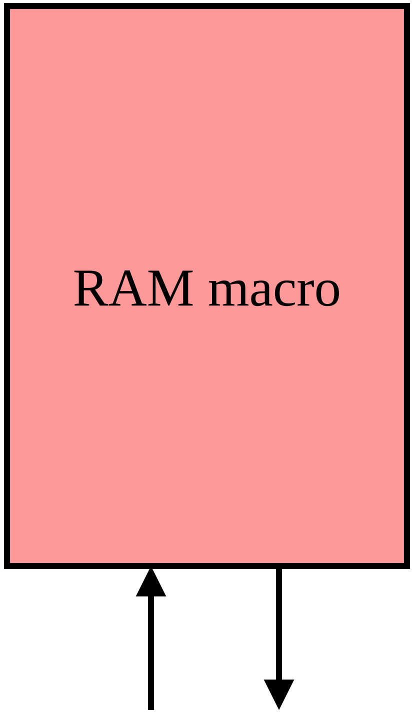
Source: *Helmut Föll*

Memory View 4: Memory Types for Custom-Designed Chips (Also known as "ASICs")

- Memories for custom processors can be built in a number of ways:
  1) On-chip “macro” memory arrays
     A. Think of as a single giant standard cell
     B. FPGAs include them (“block RAMs” or “block memory”)
  2) On-chip memory synthesized from verilog
     A. standard cells (e.g., NANDs, NORs, FFs, etc.)
     B. FPGA combinational logic blocks, LUTs, etc.
  3) Off-chip memories (often for > approx. 10 MB)
     - Very large DRAM
     - Non-volatile memory such as flash memory
     - (We could also include disks, NAS, cloud, etc.)

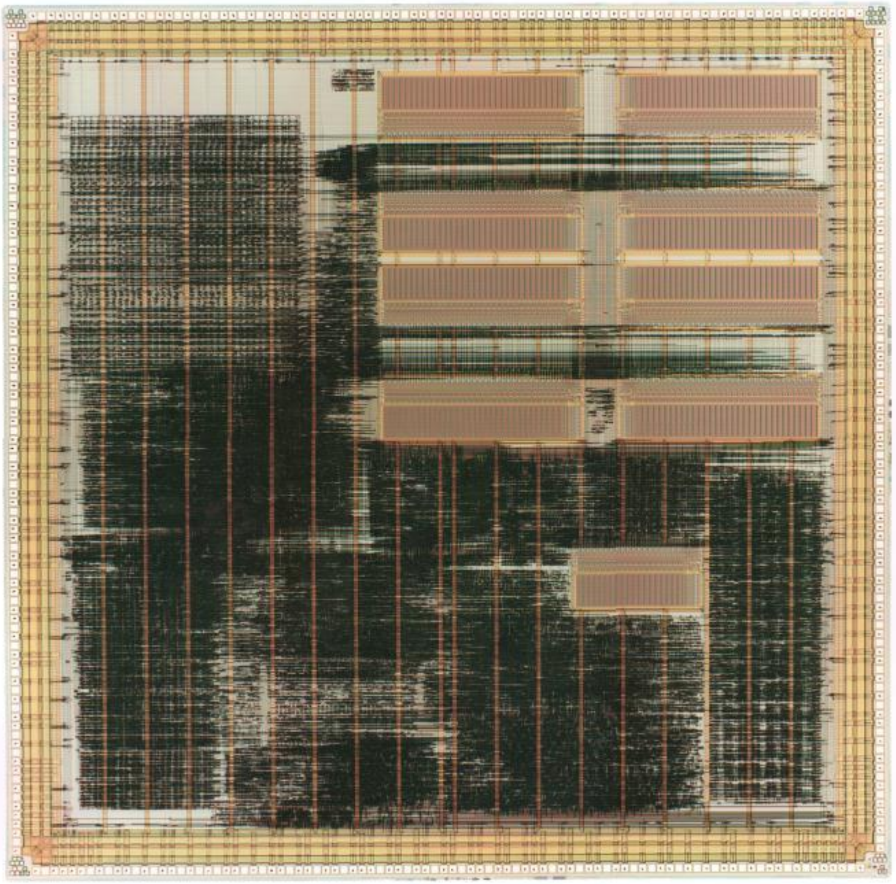
1A) On-Chip Memory "Macro" Arrays

- Memory macro-cell generators are available for larger memories
- Typically a software tool generates a large variety of possible memories where a user may select options such as:
  - Number of words
  - Word-width (in bits)
  - Number of read ports
  - Number of write ports
  - Rd/wr or ROM
  - Built-in test circuits
  - Registered inputs and/or outputs
- Tool produces models for verilog, place & route, and other CAD views



1A) On-Chip Memory "Macro" Arrays

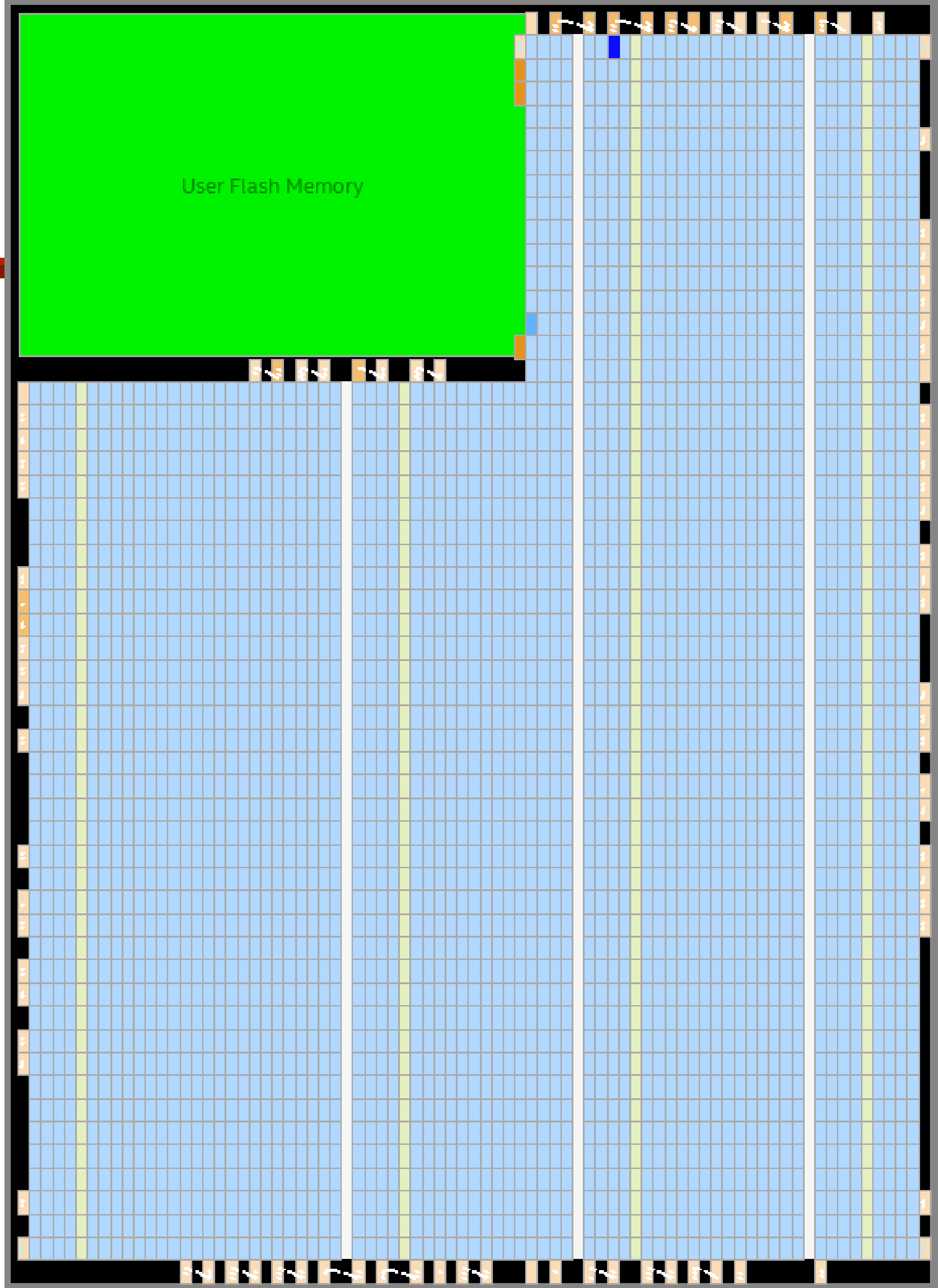
- Generally very area efficient due to dense memory cells (single-ported memories likely use 6-transistor (6T) memory cells)
- Generally good energy efficiency due to low-activity memory array architecture
- Example: CMOS chip



[T. Nanya, et al., TITAC-2 0.5 um CMOS, 496K transistors, 12.15 mm × 12.15 mm processor]

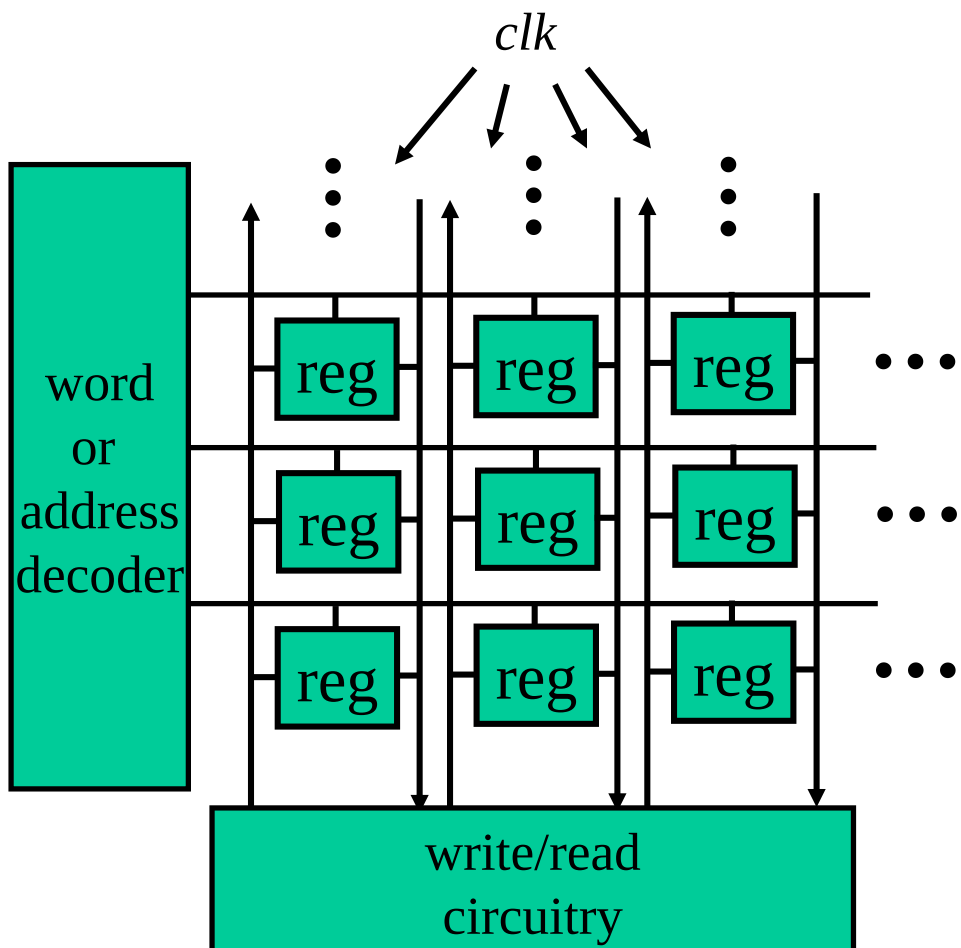
1B) On-chip Memory "macro" arrays: FPGAs

- Example: FPGA
- Altera Max 10 10M50DAF484C7G chip
- Yellow rectangles are M9K memory blocks
  - Each block contains 8192 bits (9216 including parity)
  - 182 on each chip
  - Total of 182 KBytes (204 KB)
- Light-blue rectangles: Logic Array Blocks (LAB)
- White rectangles: hardware 18x18 multipliers (144 on chip)



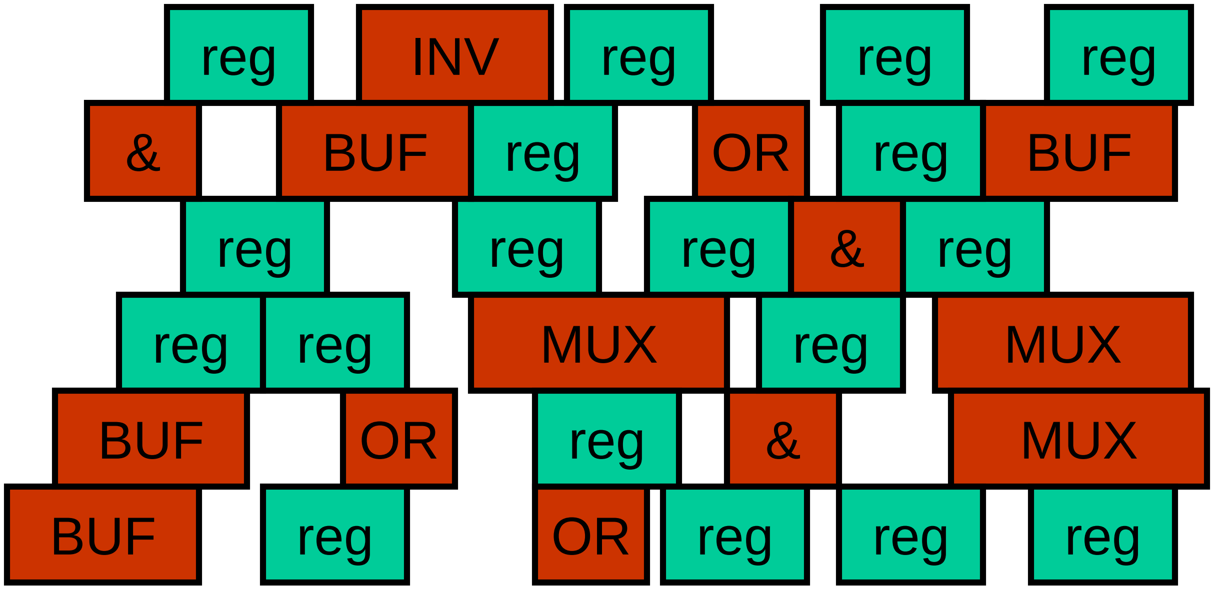
2) Synthesized Memory

- Can synthesize memory from standard cells
  - Memory cells are now flip-flops
  - *clk* likely routed to all cells
  - Probably best for small memories only
  - Read bitline logic may be muxes



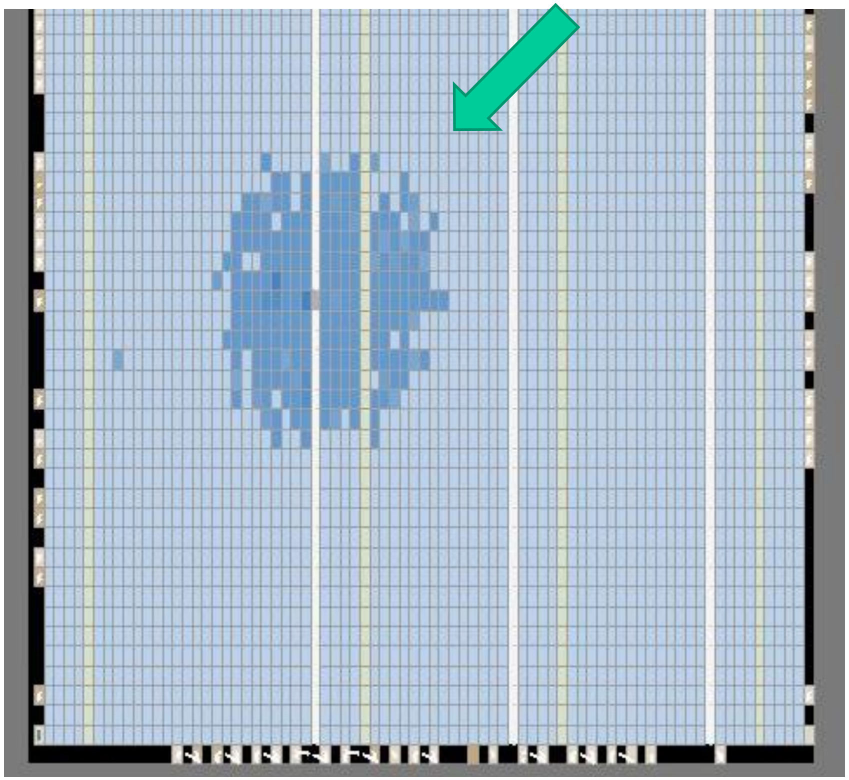
2A) Synthesized Memory: Standard Cells

- Standard cell layout is typically irregular
  - Wires not shown
  - Clocks routed to each “reg” (flip-flop)



2A) Synthesized Memory: FPGA "building block" Cells

- Our FPGA chip has 50,000 "logic element" blocks which include logic and a few memory elements and can be configured into an array memory



Verilog Memories

- Declaring a 16-bit, 128-word memory
  - `reg [15:0]   Mem [0:127];`
- Reading a memory

```
source1 = Mem[addr_rd];                  // combinational logic
```

  - This is combinational logic
  - Essentially a massive mux choosing among FF outputs
- Writing a memory

```
Mem[addr_wr] <= #1 c_datapath_out;     // makes sense for a
                                       // memory made of FFs
```

  - Writing is done in a way very much like writing FF registers
  - Remember: for this class, always use non-blocking writes

Verilog SRAM Memories—Combinational (Asynchronous) Read

- This memory performs writes on the positive edge of the clock when *write_enable* is high
- The output is not controlled by the clock and outputs the correct memory word for any address on *addr_rd*
  - Picture a large mux tree connecting every word in the memory to the output port
  - Sometimes called an "asynchronous read"

```
reg [15:0]    mem [0:127];

always @(posedge clk) begin
  if (write_enable == 1'b1) begin
    mem[addr_wr] <= #1 data_in;
  end
end

assign data_out = mem[addr_rd];
```

Verilog SRAM Memories— Synchronous Read

- This memory also performs writes on the positive edge of the clock when *write_enable* is high
- However it contains a "synchronous read" which updates the output port only on the active edge of the clock
  - There is now one clock cycle of delay from when *addr_rd* is valid to when the read output is valid
- The M9K memory blocks in Altera FPGAs work this way

```
reg [15:0]    mem [0:127];

always @(posedge clk) begin
  if (write_enable == 1'b1) begin
    mem[addr_wr] <= #1 data_in;
  end

  data_out <= #1 mem[addr_rd];
end
```

Verilog SRAM Memories— With Read Enable

- Adding a read enable capability does not change the functional usage of a memory—there is no functional issue with ignoring data that was unnecessarily read
- However enabling the read of a memory may enable a significant reduction in the power dissipation of the memory
  - Depends on the fraction of cycles that perform reads
  - Depends on the power of adding the read enable capability in the RAM and also of generating the read enable signal

```
reg [15:0]    mem [0:127];

always @(posedge clk) begin
  if (write_enable == 1'b1) begin
    mem[addr_wr] <= #1 data_in;
  end

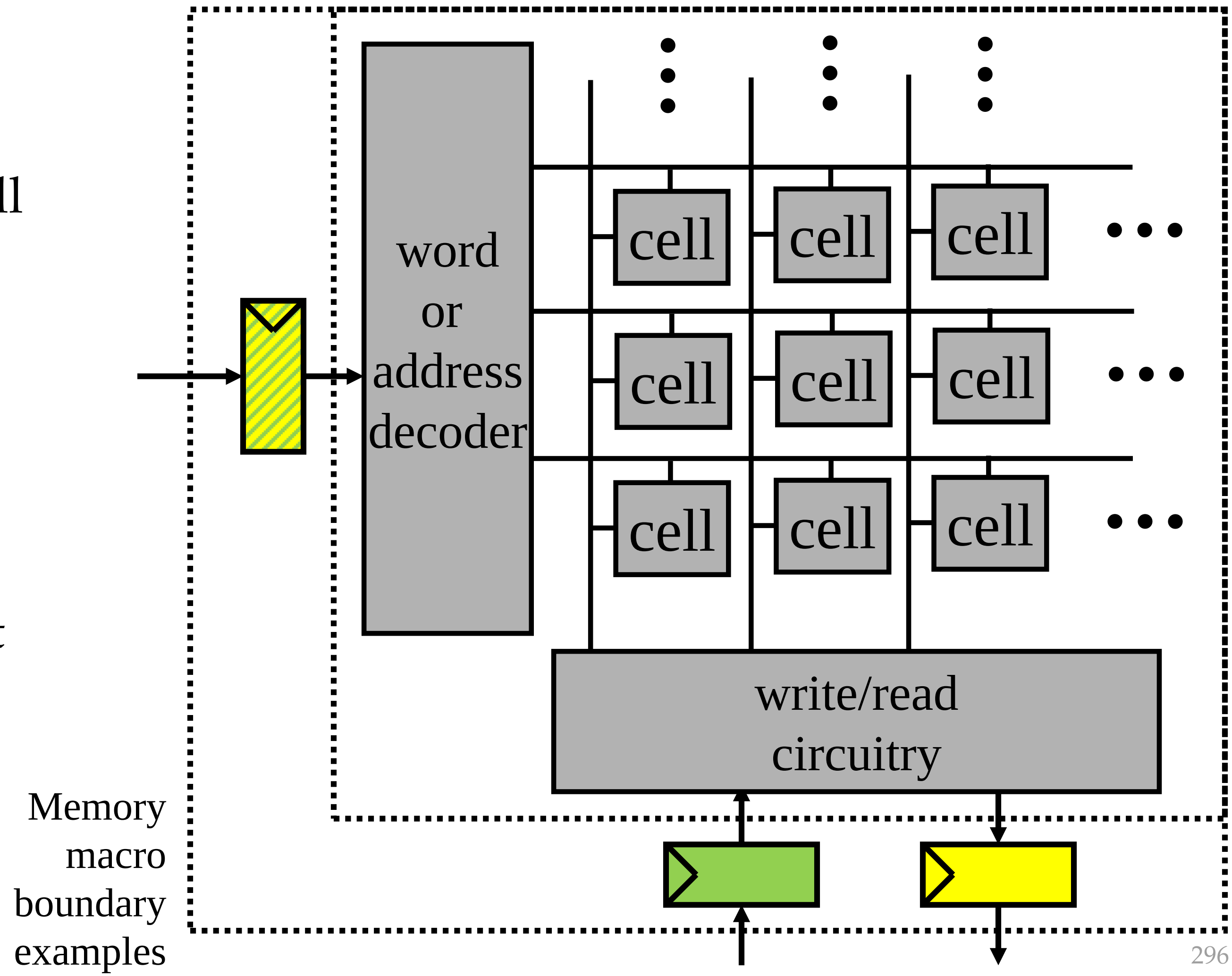
  if (read_enable == 1'b1) begin
    data_out <= #1 mem[addr_rd];
  end
end
```

Sub-Word Operations and Multiple Read Ports

- Reading and writing portions of words
  - It is not possible to access a portion of a word without first reading the whole word, in many simulators and CAD tools (though it is supported in some). Therefore, it is good practice to not do it! So don't in this class.
  - `source1 = Mem[addr_rd][5];     // won't work sometimes`
  - **`temp    = Mem[addr_rd];        // use these 2 lines instead`**
    `source1 = temp[5];`
- Multiple unclocked read ports
  - Simply make individual read statements for each read port
  - ```
    data1 = Mem[addr_rd1];
    data2 = Mem[addr_rd2];
    data3 = Mem[addr_rd3];
    ```
    ```

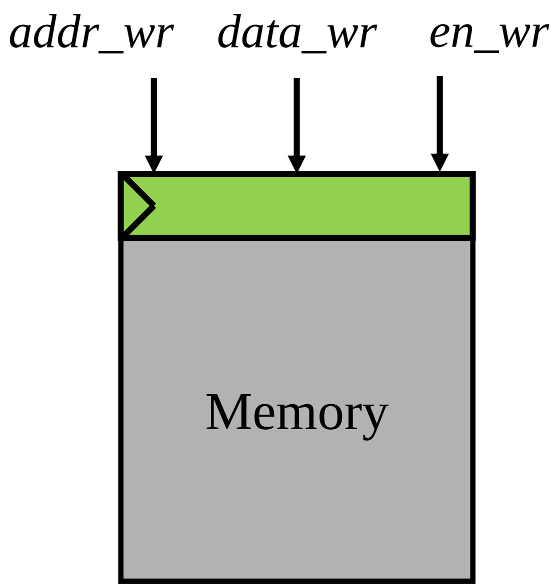
Memory Pipelining/Timing
Timing Style 1

- Style 1: Registers are outside the cell array
- Memory macros may contain registers for inputs or outputs or not at all



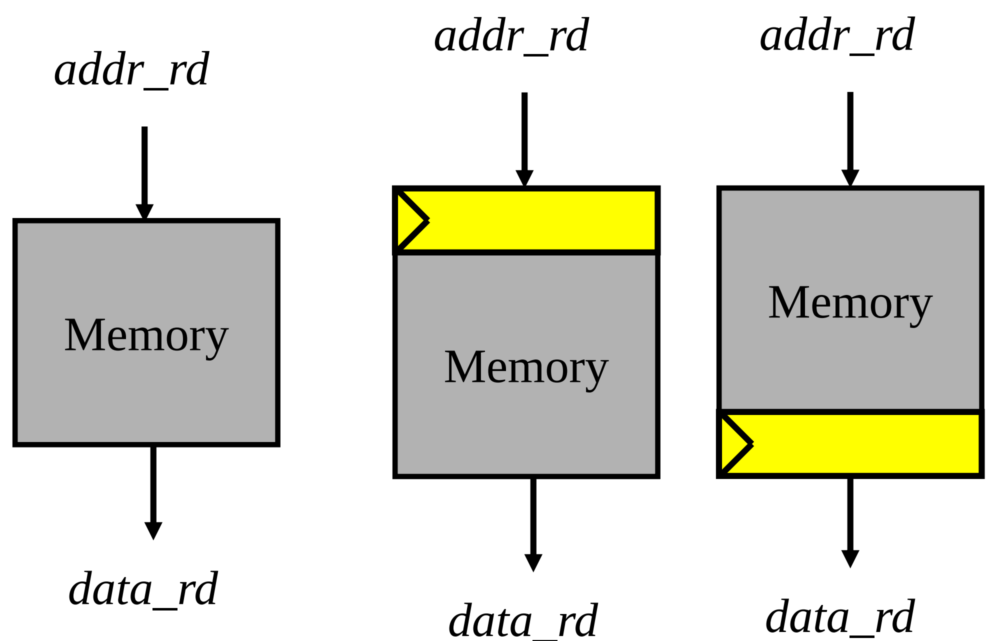
Memory Pipelining/Timing
Writes

- Synchronous-write RAMs are the most robust and common style
- Write operations complete one cycle after valid address, data, and *en_wr* = 1 are present at the inputs



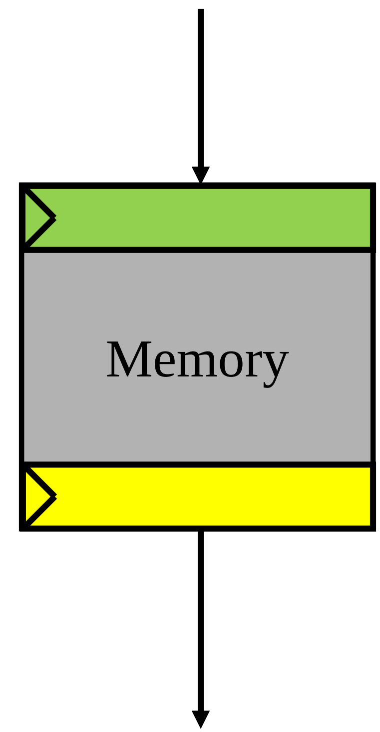
Memory Pipelining/Timing Reads

- Two main timing models are common for RAM reads: (left) *asynchronous* (straight through combinational), and (right) *synchronous* with one cycle latency from *addr_rd* to valid *data_rd*
- The two synchronous models operate the same except when considered with the timing of writes to the same address



Simultaneous Write and Read Operations

- What is the read data when a read and write are performed to the same address? Good question! It depends on the specific circuits used in the memory. There are several possibilities:
  - The "old" data existing in the previous cycle
  - The "new" data being currently written
  - "Don't care" – this means it could be old, new, or some bits from each



Simultaneous Write and Read Operations

- Example: M9K block RAMs with various port configurations and various selectable simultaneous write/read characteristics
- Selectable "old" or "new" from a single port is likely implemented by the order the RAM performs the two operations

3.5. Selecting Read-During-Write Output Choices

- Single-port RAM supports only same-port read-during-write. The clock mode must be either single clock mode or input/output clock mode.
- Simple dual-port RAM supports only mixed-port read-during-write. The clock mode must be either single clock mode, or input/output clock mode.
- True dual-port RAM supports same port read-during-write and mixed-port read-during-write:
  - For same port read-during-write, the clock mode must be either single clock mode, input/output clock mode, or independent clock mode.
  - For mixed port read-during-write, the clock mode must be either single clock mode, or input/output clock mode.

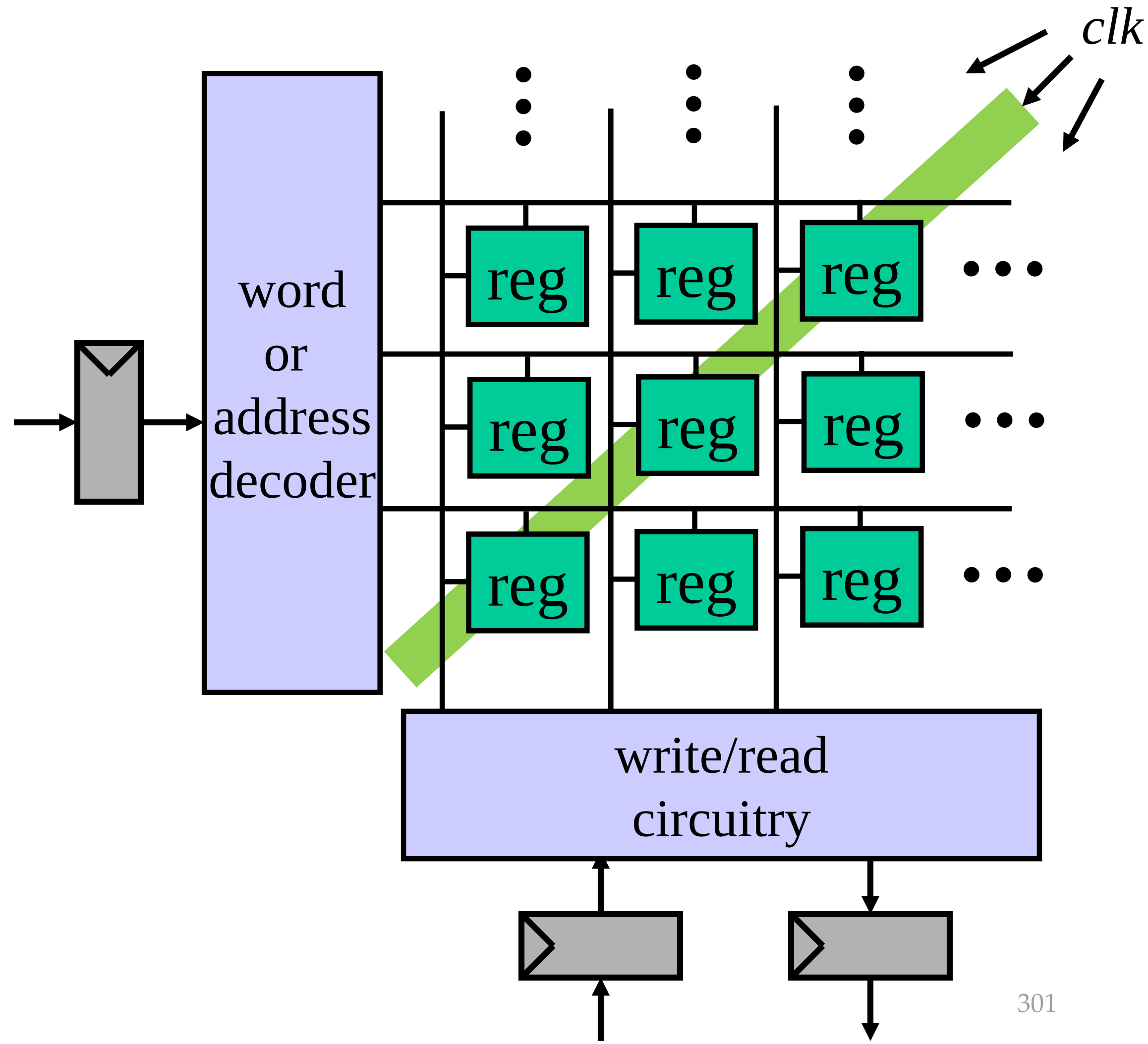
Note: If you are not concerned about the output when read-during-write occurs and want to improve performance, select **Don't Care**. Selecting **Don't Care** increases the flexibility in the type of memory block being used if you do not assign block type when you instantiate the memory block.

Table 12. Output Choices for the Same-Port and Mixed-Port Read-During-Write

Memory Block	Single-Port RAM	Simple Dual-Port RAM	True Dual-Port RAM	
	Same-Port Read-During-Write	Mixed-Port Read-During-Write	Same-Port Read-During-Write	Mixed-Port Read-During-Write
M9K	• Don't Care • New Data • Old Data	• Old Data • Don't Care	• New Data • Old Data	• Old Data • Don't Care

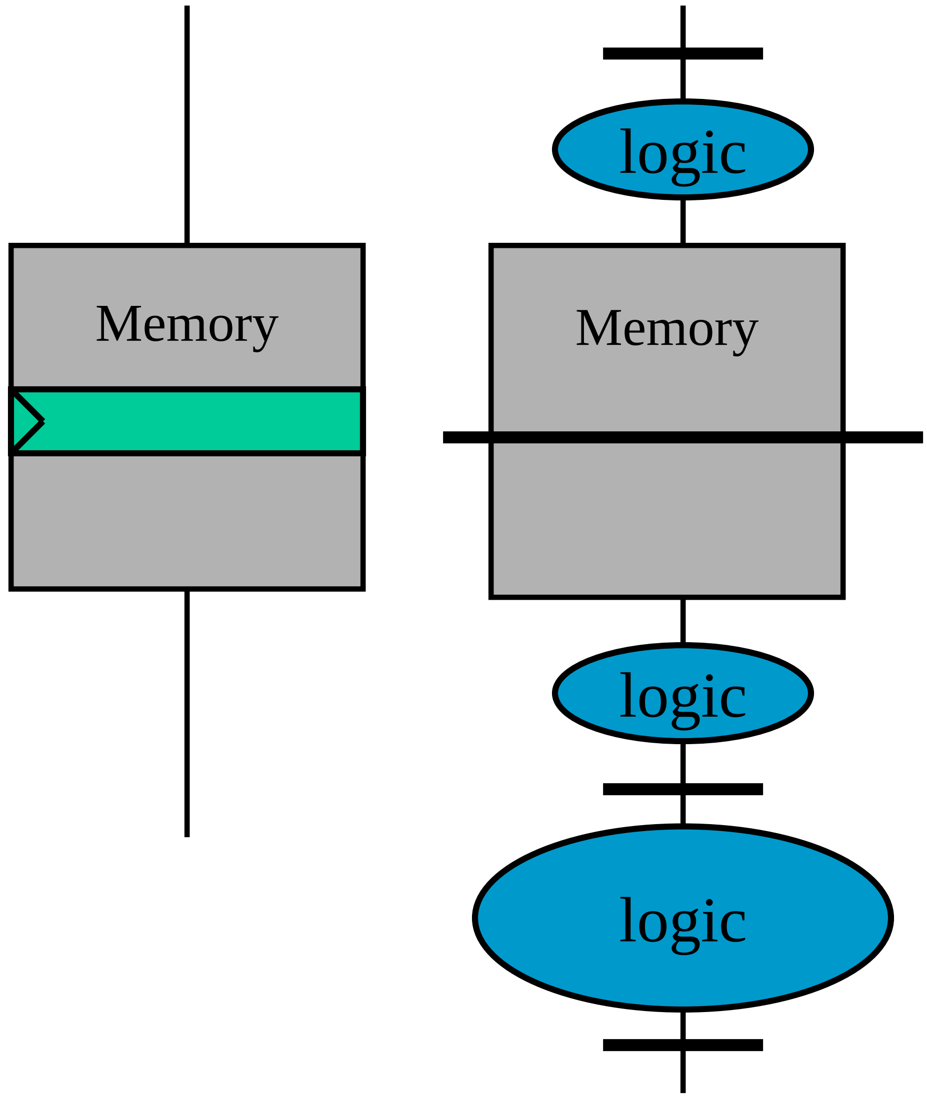
Memory Pipelining/Timing
Timing Style 2

- Style 2: Register is *in the middle* of the memory array
- Memories built with individual FFs for memory cells effectively contain a pipeline register across the middle of the entire memory array



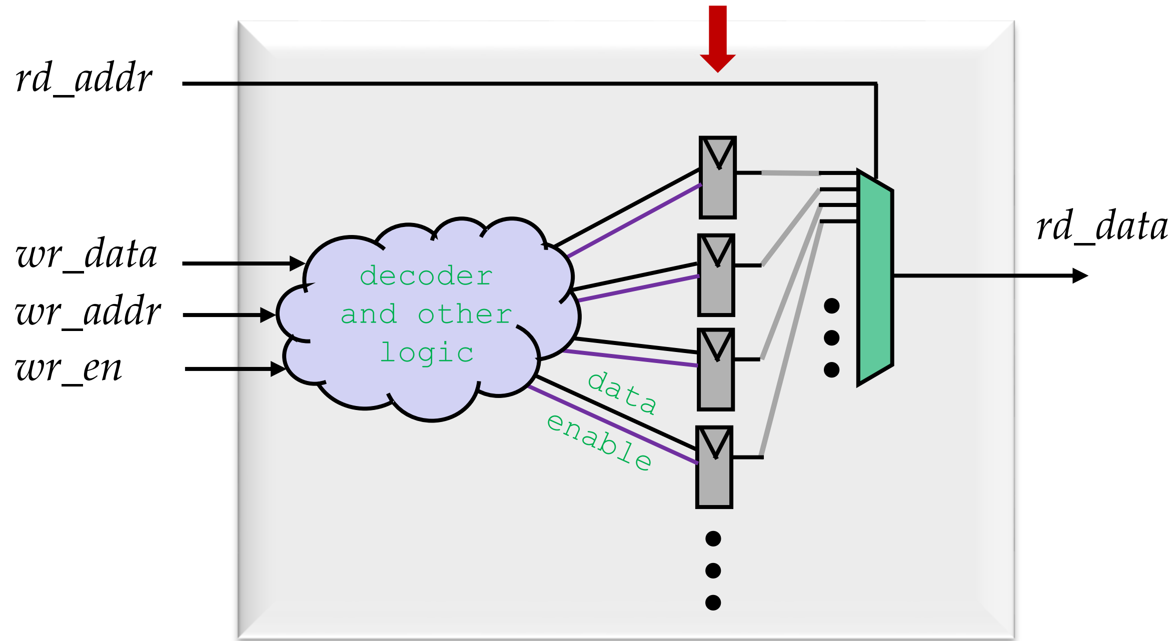
Memory Pipelining/Timing
Timing Style 2

- The built-in pipeline stage is somewhere in the middle of the memory block
- A well-balanced system would therefore place a reduced amount of logic before and after the memory array to maintain a high clock frequency



Block Diagram of a FF-based Memory with an Asynchronous (Combinational) Read Port

- There is only a combinational logic delay from the read address to the read data
- This block diagram *is not* a valid "pipelined block diagram" and would therefore be confusing to use to design a pipeline
- A simultaneous read and write to the same address will result in the read returning the *old* data, keeping in mind that the read occurs in a single cycle



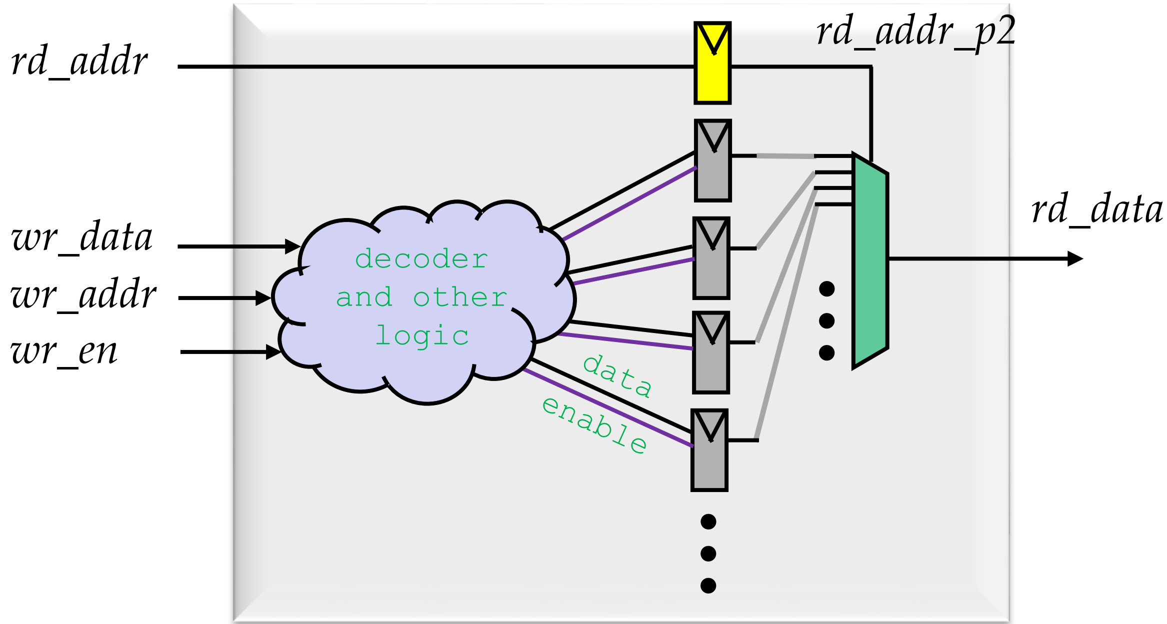

```
reg  [15:0]    mem [0:127];
wire [15:0]    rd_data;

always @(posedge clk) begin
  if (wr_en == 1'b1) begin
    mem[wr_addr] <= #1 wr_data;
  end
end

assign rd_data = mem[rd_addr];
```

Block Diagram of a FF-based Memory with a Synchronous Read Port

- There is a single-cycle delay from the read address to the read data
- This block diagram *is* a valid "pipelined block diagram" but is perhaps not the desired circuit
- A simultaneous read and write to the same address will result in the read returning the *new* data




```
reg [15:0]    mem [0:127];
reg [15:0]    rd_data;
reg [10:0]    rd_addr_p2;

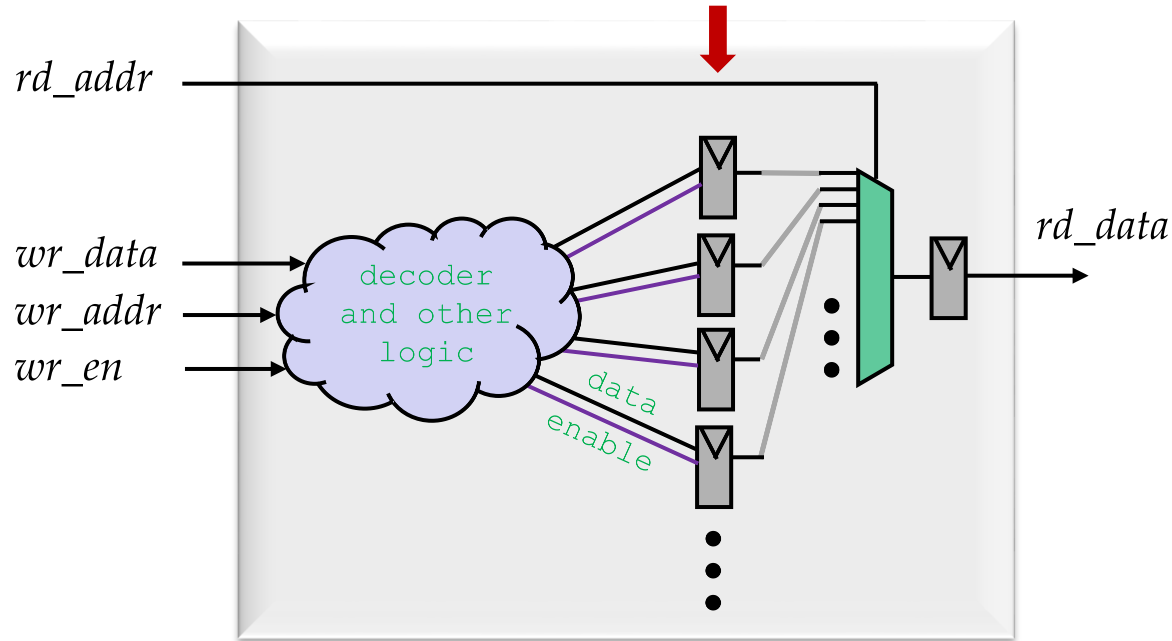
always @(posedge clk) begin
  if (wr_en == 1'b1) begin
    mem[wr_addr] <= #1 wr_data;
  end

  rd_addr_p2 <= #1 rd_addr;
end

assign rd_data = mem[rd_addr_p2];
```

Block Diagram of a FF-based Memory with a Synchronous Read Port

- There is a single-cycle delay from the read address to the read data
- A simultaneous read and write to the same address will result in the read returning the *old* data
- This diagram *is not* a valid "pipelined block diagram" and would be confusing to use to design a pipeline
  - The signal *rd_addr* crosses a vertical line of registers without being registered which violates the diagram's fundamental definition



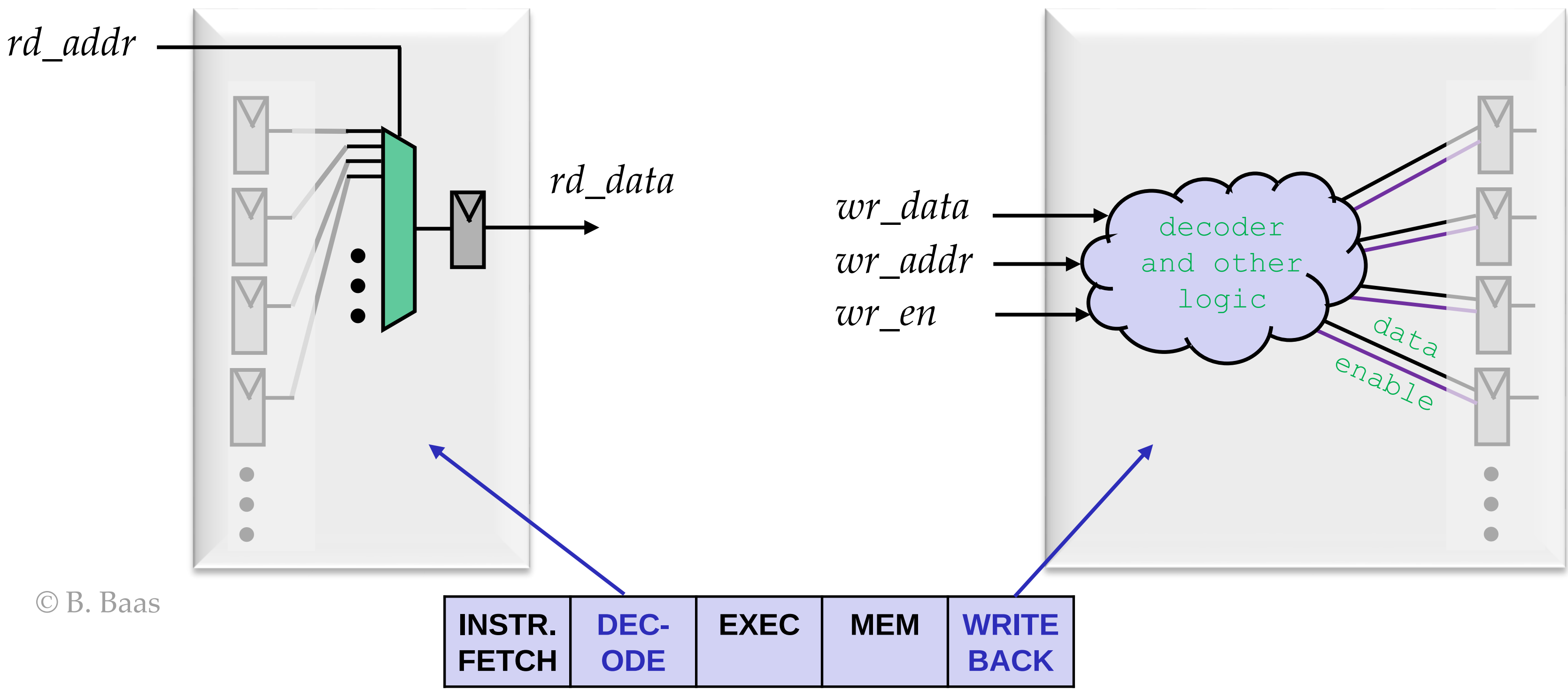

```
reg [15:0]     mem [0:127];
reg [15:0]     rd_data;

always @(posedge clk) begin
  if (wr_en == 1'b1) begin
    mem[wr_addr] <= #1 wr_data;
  end

  rd_data <= #1 mem[rd_addr];
end
```

Block Diagram of a FF-based Memory with a Synchronous Read Port

- The read and write operations are in many ways independent; they interact only through the central memory cell array
- Another reasonable approach is to split the memory block diagram into two parts and draw the two halves in different sections of the system
  - "Hazards" of interactions between writes and reads are considered separately, such as with pipeline diagrams when designing processors



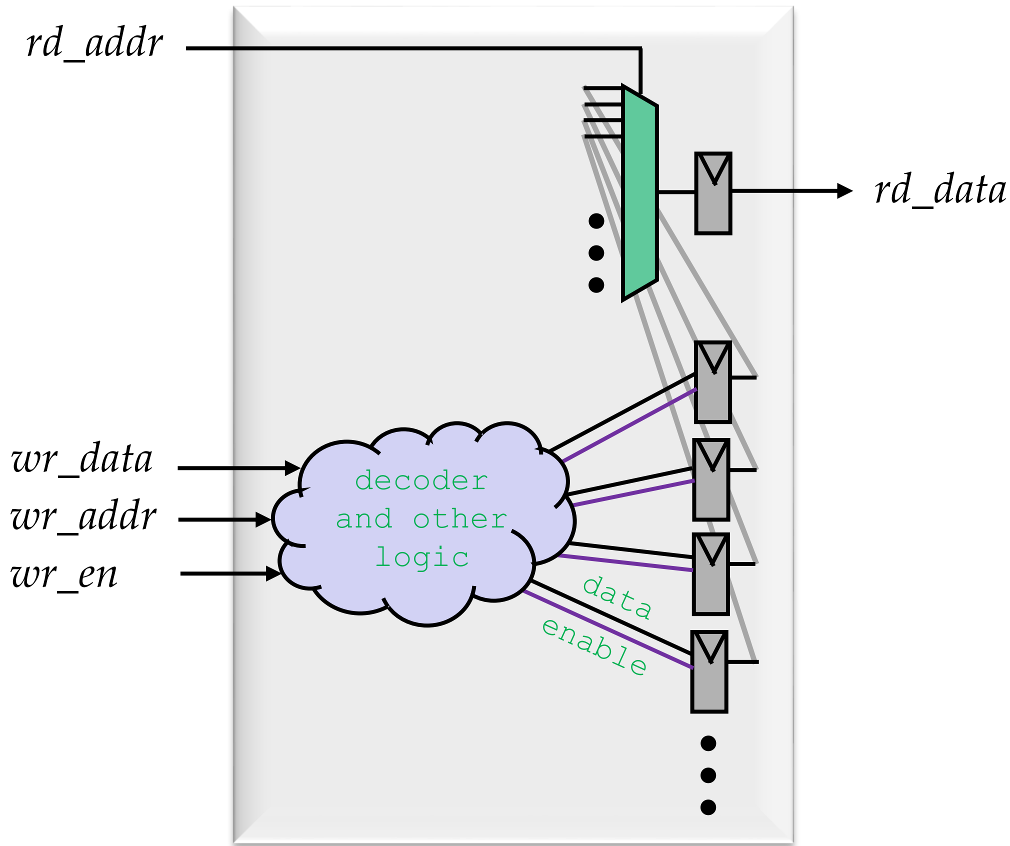
Pipelined Block Diagram of a FF-based Memory with a Synchronous Read Port

- This *is* a valid "pipelined block diagram" of a FF-based memory with a synchronous read port and is the one you should use
- Internal cell array bits (gray lines) flow right to left and are not pipelined (registered)

```
reg [15:0]    mem [0:127];
reg [15:0]    rd_data;

always @(posedge clk) begin
  if (wr_en == 1'b1) begin
    mem[wr_addr] <= #1 wr_data;
  end

  rd_data <= #1 mem[rd_addr];
end
```



ROMs – 1) Synthesized from Std Cells (A *Combinational* Circuit)

- Small ROMs can be efficiently synthesized from standard cells
- Implementations are more efficient if the data is less random in an information-theory sense
  - Ex: 10-bit *input*, *out*=1 if *input*/4 is an integer
- It is advisable to generate tables from a program such as matlab

```
// todo: add "default" case for safety
always @(input) begin
  case (input)
    4'b0000: begin  real=8'b01000000; imag=8'b00000000;  end  // angle = 0.00000
    4'b0001: begin  real=8'b00111011; imag=8'b00011000;  end  // angle = 0.12500
    4'b0010: begin  real=8'b00101101; imag=8'b00101101;  end  // angle = 0.25000
    4'b0011: begin  real=8'b00011000; imag=8'b00111011;  end  // angle = 0.37500
    4'b0100: begin  real=8'b00000000; imag=8'b01000000;  end  // angle = 0.50000
    4'b0101: begin  real=8'b11101000; imag=8'b00111011;  end  // angle = 0.62500
    4'b0110: begin  real=8'b11010011; imag=8'b00101101;  end  // angle = 0.75000
    4'b0111: begin  real=8'b11000101; imag=8'b00011000;  end  // angle = 0.87500
    4'b1000: begin  real=8'b11000000; imag=8'b00000000;  end  // angle = 1.00000
    4'b1001: begin  real=8'b11000101; imag=8'b11101000;  end  // angle = 1.12500
    4'b1010: begin  real=8'b11010011; imag=8'b11010011;  end  // angle = 1.25000
    4'b1011: begin  real=8'b11101000; imag=8'b11000101;  end  // angle = 1.37500
    4'b1100: begin  real=8'b00000000; imag=8'b11000000;  end  // angle = 1.50000
    4'b1101: begin  real=8'b00011000; imag=8'b11000101;  end  // angle = 1.62500
    4'b1110: begin  real=8'b00101101; imag=8'b11010011;  end  // angle = 1.75000
    4'b1111: begin  real=8'b00111011; imag=8'b11101000;  end  // angle = 1.87500
  endcase
end
```

- This example table has:
  - 4-bit input address
  - 16-bit (8-bit + 8-bit complex) output

ROMs – 1) Synthesized from Std Cells

- If applicable, matlab may be a good choice for a program to print the verilog table as plain text
  - You will need several versions to get it right so rapid (re)generation is a huge time saver
  - matlab has rock-solid common functions, rounding, etc.
  - matlab has superb plotting capabilities for checking all sorts of characteristics such as bias, frequency response, etc.
  - An automatically generated table is easy to adapt to other specifications such as binary word width, number format, etc. in case the problem specification changes
  - Print everything between "case" and "endcase" then copy & paste the matlab output into your verilog file

ROMs – 1) Synthesized from Std Cells

- This is the matlab code that generated the previously-shown lookup table
- Copy, Paste, Run, Change, Run!

```
% table_gen.m
% 2018/02/22 Last modified (BB)

fprintf(1, 'always @(theta) begin\n');
fprintf(1, '  case (theta)\n');

% Main loop, once for each possible input
for k=0:15
   angle = 2 * pi * k / 16;
   re    = cos(angle);
   re    = round(re * 2^6);   % scale +1 -> 64 since max is +127; then round
   im    = sin(angle);
   im    = round(im * 2^6);   % scale +1 -> 64 since max is +127; then round
   fprintf(1, '    4''b%s: begin  ', real2unsigned(k,4,0));
   fprintf(1, 'real=8''b%s; ', real2twos(re,8,0));
   fprintf(1, 'imag=8''b%s;  ', real2twos(im,8,0));
   fprintf(1, 'end  ');
   fprintf(1, '// angle = %f pi', angle/pi);
   fprintf(1, '\n');
end

fprintf(1, '  endcase\n');
fprintf(1, 'end\n');
```

Using Matlab for Lookup Table Generation

- Helpful matlab commands:

```
fprintf()
real2twos()
real2unsigned
help [matlab_command_name]
```

ROMs – 2) FPGA Block RAM

- Larger ROMs on FPGAs can make use of block RAMs whose contents can be specified with a verilog "initial" block
- Read operations are synchronous and update the output port only on the active edge of the clock
  - There is now one clock cycle of delay from when *addr_rd* is valid to when the read output is valid
- The M9K memory blocks in Altera FPGAs work this way
- This is the ***only*** case when it is ok to break our rule of no `initial` blocks in hardware verilog!

```
reg [7:0]    rom [0:127];

// Yes, this is HW verilog
initial begin
  rom[0] = 8'h35;
  rom[1] = 8'h2E;
  rom[2] = 8'hFF;
  ...
end

always @(posedge clk) begin
  data_out <= #1 rom[addr_rd];
end
```