Index of supported patterns for Yosys memory_libmap memory mapper

- These are examples of patterns. Combinations are also supported
- TODO: write a blog post about the motivation for these changes

Supported Patterns

Asynchronous-read RAM

- This will result in LUT RAM on supported targets
- Included in generate.py

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];
always @(posedge clk)
     if (write_enable)
          mem[write_addr] <= write_data;
assign read_data = mem[read_addr];
```

Synchronous SDP with clock domain crossing

- Will result in block RAM or LUT RAM depending on size
- No behavior guarantees in case of simultaneous read and write to the same address
- Included in generate.py

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge write_clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
end

always @(posedge read_clk) begin
    if (read_enable)
        read_data <= mem[read_addr];
end
```

Synchronous SDP read first

- The read and write parts can be in the same or different processes.

- Will result in block RAM or LUT RAM depending on size
- As long as the same clock is used for both, yosys will ensure read-first behavior. This may require extra circuitry on some targets for block RAM. If this is not necessary, use one of the patterns below.
- Included in generate.py
- lutram_1w1r in arch/common/lutram.v
    - Unconditional read

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
    if (read_enable)
        read_data <= mem[read_addr];
end
```

Synchronous SDP with undefined collision behavior

- Like above, but the read value is undefined when read and write ports target the same address in the same cycle
- Tested on all arch in https://github.com/YosysHQ/yosys/pull/3351

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
     if (write_enable)
     mem[write_addr] <= write_data;

     if (read_enable) begin
     read_data <= mem[read_addr];
     // 👇 this if block 👇
     if (write_enable && read_addr == write_addr)
          read_data <= 'x;
     end
end
```

- Or below, using the no_rw_check attribute

```
(* no_rw_check *)
```

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
      if (write_enable)
            mem[write_addr] <= write_data;

      if (read_enable)
            read_data <= mem[read_addr];
end
```

Synchronous SDP with write-first behavior

- Will result in block RAM or LUT RAM depending on size
- May use additional circuitry for block RAM if write-first is not natively supported. Will always use additional circuitry for LUT RAM.
- Included in generate.py

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
      if (write_enable)
            mem[write_addr] <= write_data;

      if (read_enable) begin
            read_data <= mem[read_addr];
            if (write_enable && read_addr == write_addr)
                  read_data <= write_data;
      end
end
```

Synchronous SDP with write-first behavior (alternate pattern)

- This pattern is supported for compatibility, but is much less flexible than the above
- sync_ram_sdp in arch/common/blockram.v

```
reg [ADDR_WIDTH - 1 : 0] read_addr_reg;
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
```

```
      if (write_enable)
            mem[write_addr] <= write_data;
      read_addr_reg <= read_addr;
end

assign read_data = mem[read_addr_reg];
```

Asynchronous-read single-port RAM

- Will result in single-port LUT RAM on supported targets
- Included in generate.py

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];
always @(posedge clk)
      if (write_enable)
            mem[addr] <= write_data;
assign read_data = mem[addr];
```

Synchronous single-port RAM with mutually exclusive read/write

- Will result in single-port block RAM or LUT RAM depending on size
- This is the correct pattern to infer ice40 SPRAM (with manual ram_style selection)
- On targets that don't support read/write block RAM ports (eg. ice40), will result in SDP block RAM instead
- For block RAM, will use "NO_CHANGE" mode if available
- Included in generate.py, arch/ice40/spram
    - ice40 SPRAM has no read_enable
    - using read_enable results in extra hardware logic

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
      if (write_enable)
        mem[addr] <= write_data;
      else if (read_enable)
        read_data <= mem[addr];
end
```

Synchronous single-port RAM with read-first behavior

- Will only result in single-port block RAM when read-first behavior is natively supported; otherwise, SDP RAM with additional circuitry will be used
- Many targets (Xilinx, ECP5, …) can only natively support read-first/write-first single-port RAM (or TDP RAM) where the write_enable signal implies the read_enable signal (ie. can never write without reading). The memory inference code will run a simple SAT solver on the control signals to determine if this is the case, and insert emulation circuitry if it cannot be easily proven.
- Included in generate.py

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
     if (write_enable)
         mem[addr] <= write_data;
     if (read_enable)
         read_data <= mem[addr];
end
```

Synchronous single-port RAM with write-first behavior

- Will result in single-port block RAM or LUT RAM when supported
- Block RAMs will require extra circuitry if write-first behavior not natively supported
- Included in generate.py

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
     if (write_enable)
          mem[addr] <= write_data;
     if (read_enable)
         if (write_enable)
             read_data <= write_data;
         else
             read_data <= mem[addr];
end
```

Synchronous read port with initial value

- Initial read port values can be combined with any other supported pattern
- If block RAM is used and initial read port values are not natively supported by the target, small emulation circuit will be inserted
- Included in generate.py

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];
reg [DATA_WIDTH - 1 : 0] read_data;
initial read_data = ‘h1234;

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
    if (read_enable)
        read_data <= mem[read_addr];
end
```

Synchronous read port with synchronous reset (reset priority over enable)

- Synchronous resets can be combined with any other supported pattern (except that synchronous reset and asynchronous reset cannot be used on a single read port)
- If block RAM is used and synchronous resets are not natively supported by the target, small emulation circuit will be inserted
- Included in generate.py

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;

    if (read_reset)
        read_data <= 'h1234;
    else if (read_enable)
        read_data <= mem[read_addr];
end
```

Synchronous read port with synchronous reset (enable priority over reset)

- Synchronous resets can be combined with any other supported pattern (except that synchronous reset and asynchronous reset cannot be used on a single read port)
- If block RAM is used and synchronous resets are not natively supported by the target, small emulation circuit will be inserted
- Included in generate.py

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
    if (read_enable)
        if (read_reset)
            read_data <= 'h1234;
        else
            read_data <= mem[read_addr];
end
```

Synchronous read port with asynchronous reset

- Asynchronous resets can be combined with any other supported pattern (except that synchronous reset and asynchronous reset cannot be used on a single read port)
- If block RAM is used and asynchronous resets are not natively supported by the target, small emulation circuit will be inserted
- Included in generate.py

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
end

always @(posedge clk, posedge read_reset) begin
    if (read_reset)
        read_data <= 'h1234;
    else if (read_enable)
```

```
        read_data <= mem[read_addr];
end
```

Initial data

- Most FPGA targets support initializing all kinds of memory to user-provided values
- If explicit initialization is not used, initial memory value is undefined
- Initial data can be provided by either initial statements writing memory cells one by one or $readmemh/$readmemb system tasks
- Included in generate.py
- $readmemh in arch/ecp5/bug1836

Write port with byte enables

- Byte enables can be used with any supported pattern
- To ensure that multiple writes will be merged into one port, they need to have disjoint bit ranges, have the same address, and the same clock
- Any write enable granularity will be accepted (down to per-bit write enables), but using smaller granularity than natively supported by the target is very likely to be inefficient (eg. using 4-bit bytes on ECP5 will result in either padding the bytes with 5 dummy bits to native 9-bit units or splitting the RAM into two block RAMs)
- Included in generate.py

```
reg [31 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
    if (write_enable[0])
        mem[write_addr][7:0] <= write_data[7:0];
    if (write_enable[1])
        mem[write_addr][15:8] <= write_data[15:8];
    if (write_enable[2])
        mem[write_addr][23:16] <= write_data[23:16];
    if (write_enable[3])
        mem[write_addr][31:24] <= write_data[31:24];
    if (read_enable)
        read_data <= mem[read_addr];
end
```

Asymmetric memory — general notes

- Included in arch/xilinx/asym_ram_sdp*

To construct an asymmetric memory (memory with read/write ports of differing widths):

- Declare the memory with the width of the narrowest intended port
- Split all wide ports into multiple narrow ports
- To ensure the wide ports will be correctly merged:
  - For the address, use a concatenation of actual address in the high bits and a constant in the low bits
  - Ensure the actual address is identical for all ports belonging to the wide port
  - Ensure that clock is identical
  - For read ports, ensure that enable/reset signals are identical (for write ports, the enable signal may vary — this will result in using the byte enable functionality)
- Asymmetric memory is supported on all targets, but may require emulation circuitry where not natively supported
- Note: when the memory is larger than the underlying block RAM primitive, hardware asymmetric memory support is likely not to be used even if present, as this is cheaper

Asymmetric memory with wide synchronous read port

- Included in generate.py

```
reg [7:0] mem [0:255];
wire [7:0] write_addr;
wire [5:0] read_addr;
wire [7:0] write_data;
reg [31:0] read_data;

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
    if (read_enable) begin
        read_data[7:0] <= mem[{read_addr, 2'b00}];
        read_data[15:8] <= mem[{read_addr, 2'b01}];
        read_data[23:16] <= mem[{read_addr, 2'b10}];
        read_data[31:24] <= mem[{read_addr, 2'b11}];
    end
end
```

Wide asynchronous read port

- Note: the only target natively supporting this pattern is Xilinx UltraScale
- Included in generate.py

```
reg [7:0] mem [0:511];
wire [8:0] write_addr;
wire [5:0] read_addr;
wire [7:0] write_data;
wire [63:0] read_data;

always @(posedge clk) begin
    if (write_enable)
        mem[write_addr] <= write_data;
end

assign read_data[7:0] = mem[{read_addr, 3'b000}];
assign read_data[15:8] = mem[{read_addr, 3'b001}];
assign read_data[23:16] = mem[{read_addr, 3'b010}];
assign read_data[31:24] = mem[{read_addr, 3'b011}];
assign read_data[39:32] = mem[{read_addr, 3'b100}];
assign read_data[47:40] = mem[{read_addr, 3'b101}];
assign read_data[55:48] = mem[{read_addr, 3'b110}];
assign read_data[63:56] = mem[{read_addr, 3'b111}];
```

Wide write port

- Included in generate.py

```
reg [7:0] mem [0:255];
wire [5:0] write_addr;
wire [7:0] read_addr;
wire [31:0] write_data;
reg [7:0] read_data;

always @(posedge clk) begin
    if (write_enable[0])
        mem[{write_addr, 2'b00}] <= write_data[7:0];
    if (write_enable[1])
        mem[{write_addr, 2'b01}] <= write_data[15:8];
    if (write_enable[2])
```

```
        mem[{write_addr, 2'b10}] <= write_data[23:16];
    if (write_enable[3])
        mem[{write_addr, 2'b11}] <= write_data[31:24];
    if (read_enable)
        read_data <= mem[read_addr];
end
```

True dual port memory — general notes

- Many different variations of true dual port memory can be created by combining two single-port RAM patterns on the same memory
- When TDP memory is used, memory inference code has much less maneuver room to create requested semantics compared to individual single-port patterns (which can end up lowered to SDP memory where necessary) — supported patterns depend strongly on the target
- In particular, when both ports have the same clock, it's likely that "undefined collision" mode needs to be manually selected to enable TDP memory inference
- The examples below are non-exhaustive — many more combinations of port types are possible
- Note: if two write ports are in the same process, this defines a priority relation between them (if both ports are active in the same clock, the later one wins). On almost all targets, this will result in a bit of extra circuitry to ensure the priority semantics. If this is not what you want, put them in separate processes.
    - Priority is **not** supported when using the verific front end and any priority semantics are ignored.
- Minimal testing of patterns in generate.py

True Dual Port — different clocks, exclusive read/write

- sync_ram_tdp in arch/common/blockram.v
    - No read_enable
- Included in arch/ecp5/memories

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk_a) begin
     if (write_enable_a)
       mem[addr_a] <= write_data_a;
     else if (read_enable_a)
       read_data_a <= mem[addr_a];
```

```
end

always @(posedge clk_b) begin
      if (write_enable_b)
         mem[addr_b] <= write_data_b;
      else if (read_enable_b)
         read_data_b <= mem[addr_b];
end
```

True Dual Port — same clock, read-first behavior

- This requires hardware inter-port read-first behavior, and will only work on some targets (Xilinx, Nexus)
- Included in arch/xilinx/priority_memory

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];

always @(posedge clk) begin
      if (write_enable_a)
         mem[addr_a] <= write_data_a;
      if (read_enable_a)
         read_data_a <= mem[addr_a];
end
always @(posedge clk) begin
      if (write_enable_b)
         mem[addr_b] <= write_data_b;
      if (read_enable_b)
         read_data_b <= mem[addr_b];
end
```

Multiple read ports

- The combination of a single write port with an arbitrary amount of read ports is supported on all targets — if a multi-read port primitive is available (like Xilinx RAM64M), it'll be used as appropriate. Otherwise, the memory will be automatically split into multiple primitives.
- Included in generate.py
- lutram_1w3r in arch/common/lutram.v
    - synchronous read, read-first

```
reg [31:0] mem [0:31];

always @(posedge clk) begin
     if (write_enable)
        mem[write_addr] <= write_data;
end

assign read_data_a = mem[read_addr_a];
assign read_data_b = mem[read_addr_b];
assign read_data_c = mem[read_addr_c];
```

Memory kind selection

The memory inference code will automatically pick target memory primitive based on memory geometry and features used. Depending on the target, there can be up to four memory primitive classes available for selection:

- FF RAM (aka logic): no hardware primitive used, memory lowered to a bunch of FFs and multiplexers
    - Can handle arbitrary number of write ports, as long as all write ports are in the same clock domain
    - Can handle arbitrary number and kind of read ports
- LUT RAM (aka distributed RAM): uses LUT storage as RAM
    - Supported on most FPGAs (with notable exception of ice40)
    - Usually has one synchronous write port, one or more asynchronous read ports
    - Small
    - Will never be used for ROMs (lowering to plain LUTs is always better)
- Block RAM: dedicated memory tiles
    - Supported on basically all FPGAs
    - Supports only synchronous reads
    - Two ports with separate clocks
    - Usually supports true dual port (with notable exception of ice40 that only supports SDP)
    - Usually supports asymmetric memories and per-byte write enables
    - Several kilobits in size
- Huge RAM:
    - Only supported on several targets:
        - Some Xilinx UltraScale devices (UltraRAM)
            - Two ports, both with mutually exclusive synchronous read and write
            - Single clock

            - Initial data must be all-0
        - Some ice40 devices (SPRAM)
            - Single port with mutually exclusive synchronous read and write
            - Does not support initial data
        - Nexus (large RAM)
            - Two ports, both with mutually exclusive synchronous read and write
            - Single clock
    - Will not be automatically selected by memory inference code, needs explicit opt-in via ram_style attribute

In general, you can expect the automatic selection process to work roughly like this:

- If any read port is asynchronous, only LUT RAM (or FF RAM) can be used.
- If there is more than one write port, only block RAM can be used, and this needs to be a hardware-supported true dual port pattern
    - … unless all write ports are in the same clock domain, in which case FF RAM can also be used, but this is generally not what you want for anything but really small memories
- Otherwise, either FF RAM, LUT RAM, or block RAM will be used, depending on memory size

This process can be overridden by attaching a ram_style attribute to the memory:

- (* ram_style = “logic” *) selects FF RAM
- (* ram_style = “distributed” *) selects LUT RAM
- (* ram_style = “block” *) selects block RAM
- (* ram_style = “huge” *) selects huge RAM

It is an error if this override cannot be realized for the given target.

Many alternate spellings of the attribute are also accepted, for compatibility with other software.

- Needs testing

Patterns only supported with Verific

Synchronous SDP with write-first behavior via blocking assignments

- Would require modifications to the Yosys Verilog frontend.
- Use this pattern instead

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];
```

```
always @(posedge clk) begin
      if (write_enable)
            mem[write_addr] = write_data;

      if (read_enable)
            read_data <= mem[read_addr];
end
```

Asymmetric memories via part selection

- Would require major changes to the Verilog frontend.
- Build wide ports out of narrow ports instead (see Asymmetric memory with wide synchronous read port)
- CHECK ME: this may already be supported with the Verific frontend?
    - It does

```
reg [31:0] mem [2**ADDR_WIDTH - 1 : 0];

wire [1:0] byte_lane;
wire [7:0] write_data;

always @(posedge clk) begin
      if (write_enable)
            mem[write_addr][byte_lane * 8 +: 8] <= write_data;

      if (read_enable)
            read_data <= mem[read_addr];
end
```

Undesired Patterns

Asynchronous writes

- Not supported in modern FPGAs
- Not supported in yosys code anyhow

```
reg [DATA_WIDTH - 1 : 0] mem [2**ADDR_WIDTH - 1 : 0];
```

```
always @* begin
      if (write_enable)
            mem[write_addr] = write_data;
end

assign read_data = mem[read_addr];
```

(Triage - Patterns that need testing)

- Please let us know on slack if you have any code that doesn't match the patterns in this document!