EECS150 - Digital Design

Lecture 11 - Project Description, Part 2: CPU Pipeline, Memory Blocks

Feb 22, 2010

John Wawrzynek

Project Overview

A. Serial Interface
B. MIPS150 pipeline structure
C. Memories, project memories and FPGAs
D. Dynamic Memory (DRAM)
E. Caches
F. Video subsystem
G. Ethernet Interface
H. Project specification and grading standard

MIPS150 Pipeline

The blocks in the datapath with the greatest delay are: IMEM, ALU, and DMEM. Allocate one pipeline stage to each:

I	X	M

Use PC register as address to IMEM and retrieve next instruction. Instruction gets stored in a pipeline register, also called "instruction register", in this case.

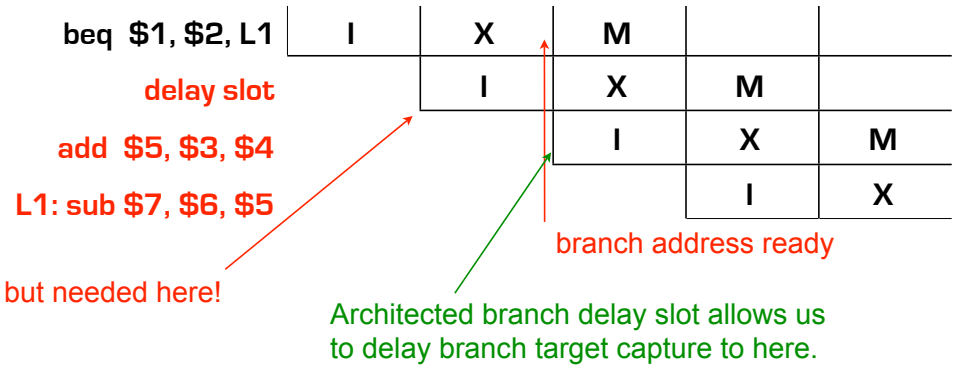
Use ALU to compute result, memory address, or compare registers for branch.

Access data memory or I/O device for load or store. Allow for setup time for register file write.

Most details you will need to work out for yourself. Some details to follow ... In particular, let's look at hazards.

MIPS 3-stage Pipeline

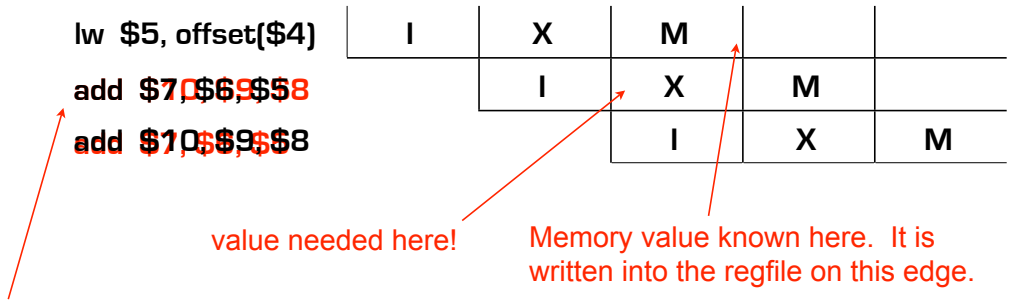
Control Hazard Example



Therefore no extra logic is required.

MIPS 3-stage Pipeline

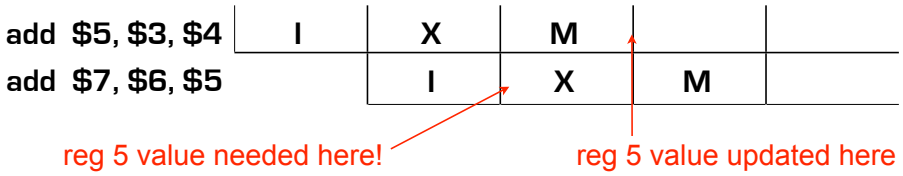
Load Hazard



"Architected load delay slot" on MIPS allows compiler to deal with the delay. No regfile bypassing needed here assuming regfile "write before read".

MIPS 3-stage Pipeline

Data Hazard



Ways to fix:

1. Stall the pipeline behind first add to wait for result to appear in register file. NOT ALLOWED this semester.
2. Selectively forward ALU result back to input of ALU.

- Need to add mux at input to ALU, add control logic to sense when to activate. A bit complex to design. Check book for details.

Project CPU Pipelining Summary

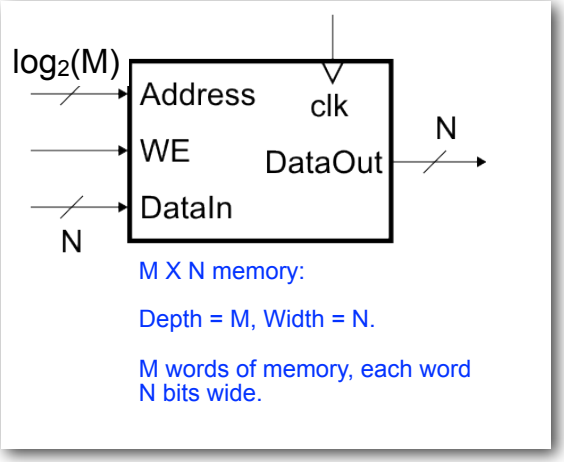
3-stage pipeline	I	X	M
	instruction fetch	execute	access data memory

- Pipeline rules:
  - Writes/reads to/from DMem use leading edge of "M"
  - Writes to RegFile use trailing edge of "M"
  - Instruction Decode and Register File access is up to you.
- 1 Load Delay Slot, 1 Branch Delay Slot
  - No Stalling may be used to accommodate pipeline hazards (in final version).
- Other:
  - Target frequency to be announced later (50-100MHz)
  - Minimize cost
  - Posedge clocking only

Memory-Block Basics

- Uses:

  Whenever a large collection of state elements is required.
  - data & program storage
  - general purpose registers
  - data buffering
  - table lookups
  - CL implementation



- Basic Types:
  - RAM - random access memory
  - ROM - read only memory
  - EPROM, FLASH - electrically programmable read only memory

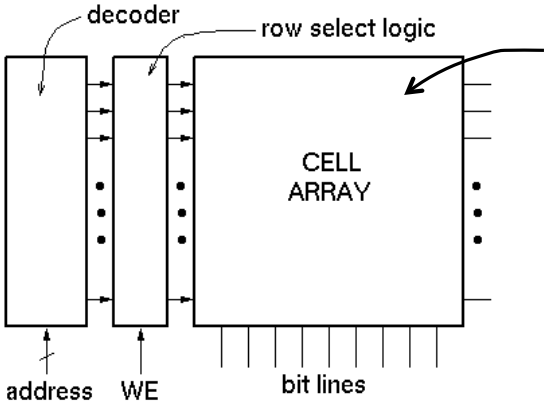
Memory Components Types:

- Volatile:
  - Random Access Memory (RAM):
    - DRAM "dynamic" Focus Monday
    - SRAM "static" Focus Today

- Non-volatile:
  - Read Only Memory (ROM):
    - Mask ROM "mask programmable"
    - EPROM "electrically programmable"
    - EEPROM "erasable electrically programmable"
    - FLASH memory - similar to EEPROM with programmer integrated on chip

All these types are available as stand alone chips or as blocks in other chips.

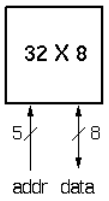
Standard Internal Memory Organization

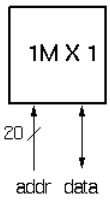


2-D arrary of bit cells. Each cell stores one bit of data.

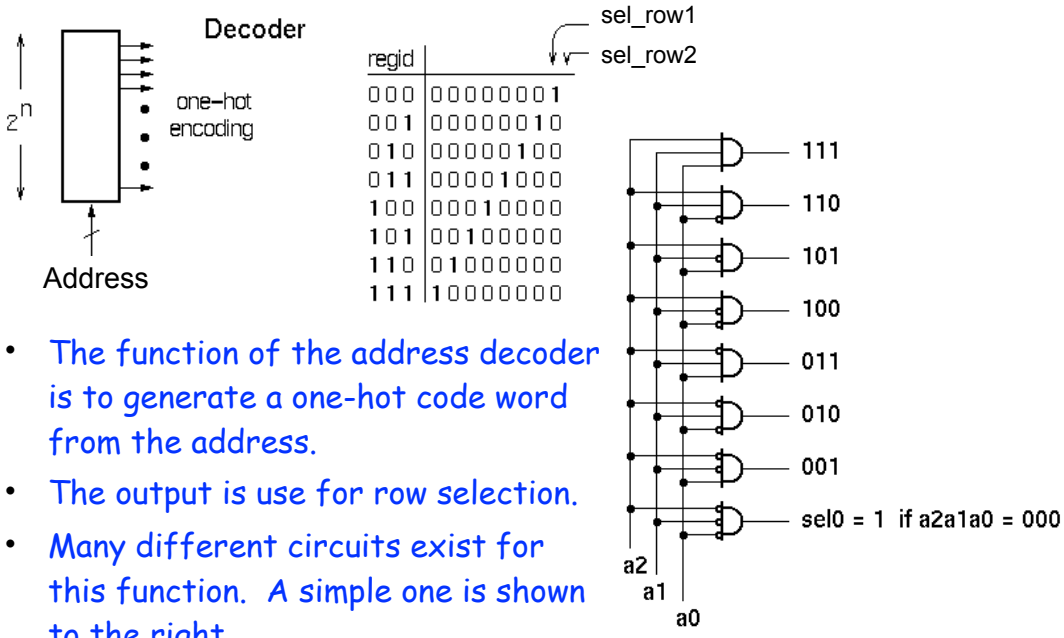
Special circuit tricks are used for the cell array to improve storage density.

- RAM/ROM naming convention:
  - examples: 32 X 8, "32 by 8" => 32 8-bit words
  - 1M X 1, "1 meg by 1" => 1M 1-bit words





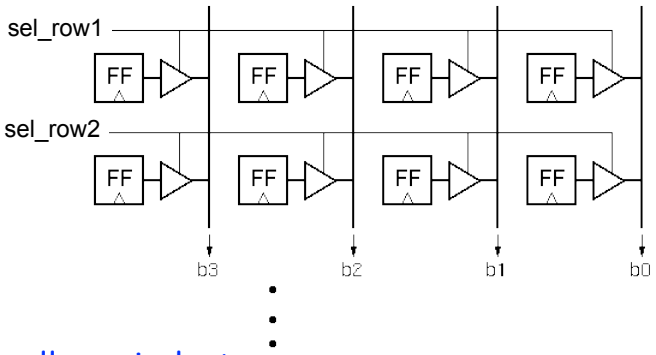
Address Decoding



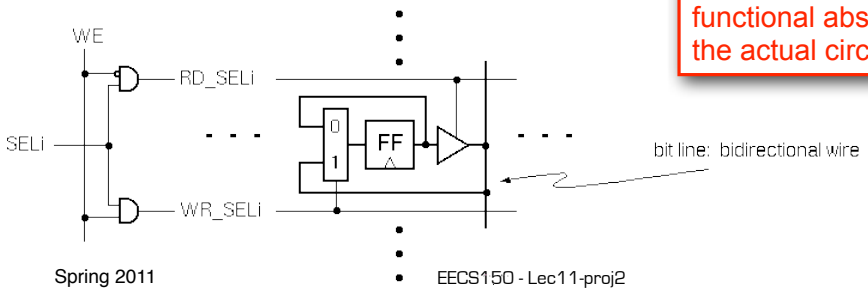
- The function of the address decoder is to generate a one-hot code word from the address.
- The output is use for row selection.
- Many different circuits exist for this function. A simple one is shown to the right.

Memory Block Internals

For read operation, functionally the memory is equivalent to a 2-D array off flip-flops with tristate outputs on each:

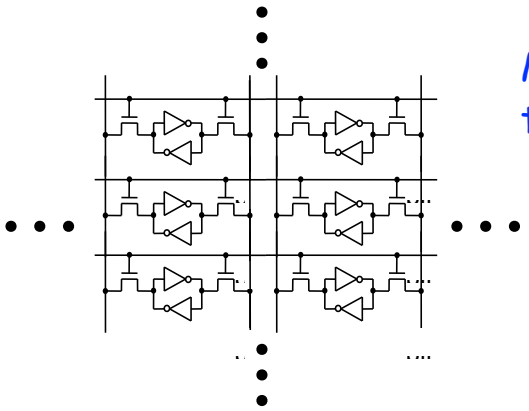


For write operation, functionally equivalent includes a means to change state value:



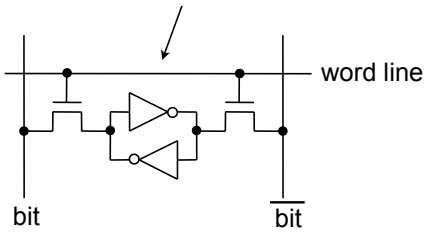
These circuits are just functional abstractions of the actual circuits used.

SRAM Cell Array Details



Most common is 6-transistor (6T) cell array.

Word selects this cell, and all others in a row.

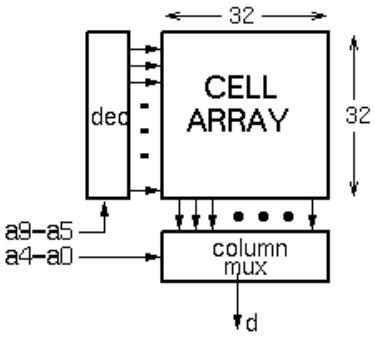


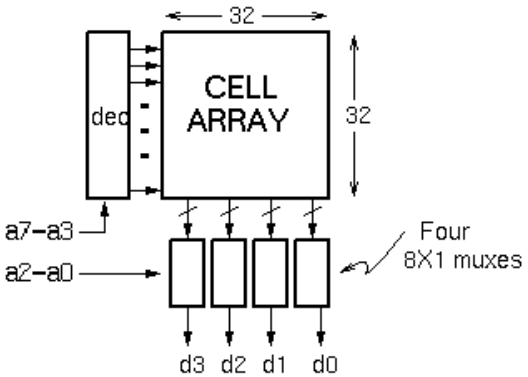
For write operation, column bit lines are driven differentially (0 on one, 1 on the other). Values overwrites cell state.

For read operation, column bit lines are equalized (set to same voltage), then released. Cell pulls down one bit line or the other.

Column MUX in ROMs and RAMs:

- Permits input/output data widths different from row width.
- Controls physical *aspect ratio*
  - Important for physical layout and to control delay on wires.



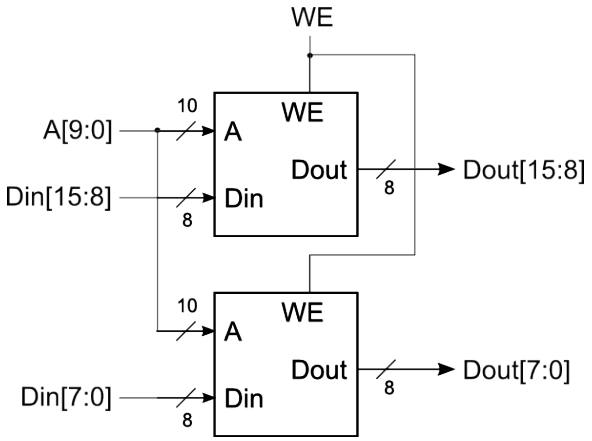


Technique illustrated for read operation. Similar approach for write.

Cascading Memory-Blocks

How to make larger memory blocks out of smaller ones.

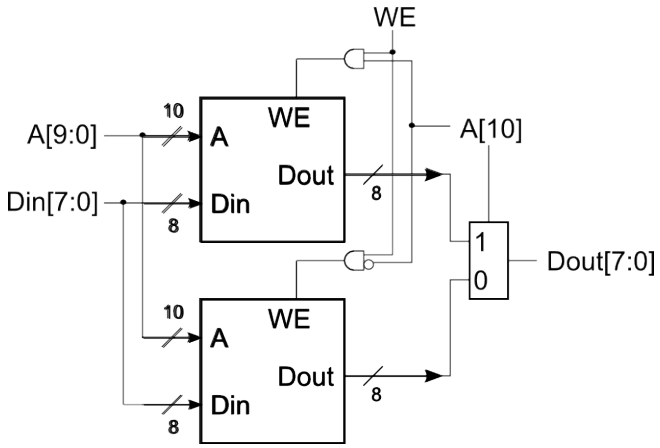
Increasing the width. Example: given 1Kx8, want 1Kx16



Cascading Memory-Blocks

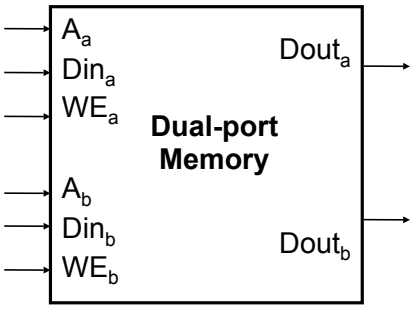
How to make larger memory blocks out of smaller ones.

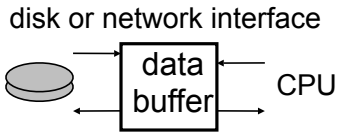
Increasing the depth. Example: given 1Kx8, want 2Kx8



Multi-ported Memory

- Motivation:
  - Consider CPU core register file:
    - 1 read or write per cycle limits processor performance.
    - Complicates pipelining. Difficult for different instructions to simultaneously read or write regfile.
    - Common arrangement in pipelined CPUs is 2 read ports and 1 write port.
  - I/O data buffering:

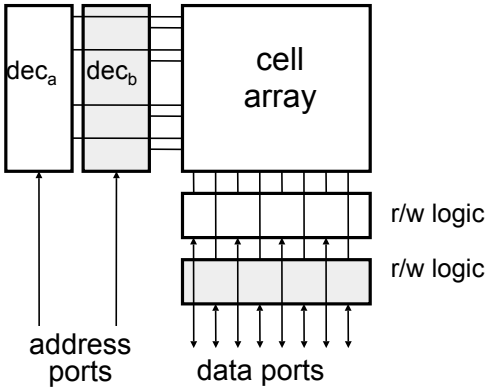




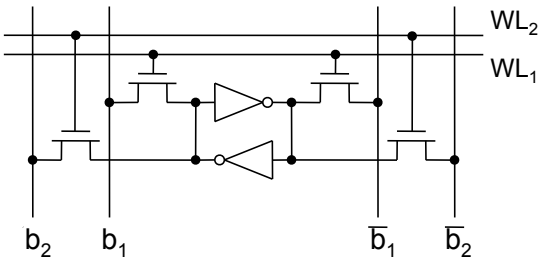
- dual-porting allows both sides to simultaneously access memory at full bandwidth.

Dual-ported Memory Internals

- Add decoder, another set of read/write logic, bits lines, word lines:



- Example cell: SRAM

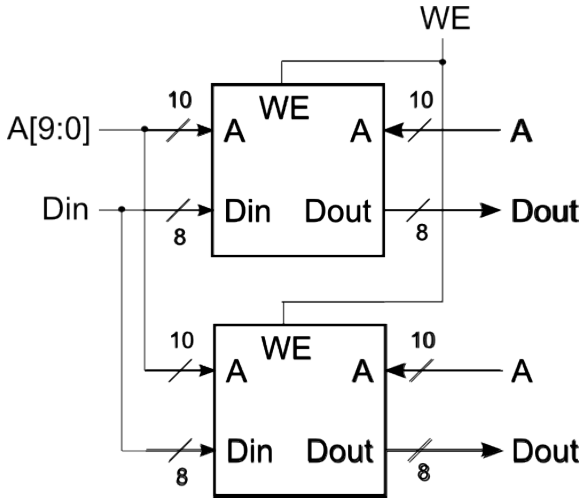


- Repeat everything but cross-coupled inverters.
- This scheme extends up to a couple more ports, then need to add additional transistors.

Adding Ports to Primitive Memory Blocks

Adding a read port to a simple dual port (SDP) memory.

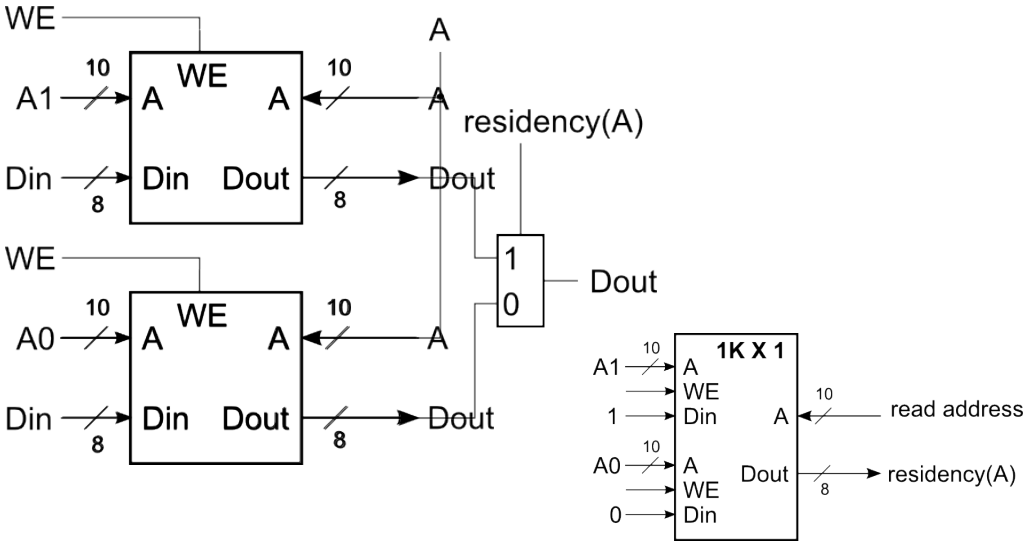
Example: given 1Kx8 SDP, want 1 write & 2 read ports.



Adding Ports to Primitive Memory Blocks

How to add a write port to a simple dual port memory.

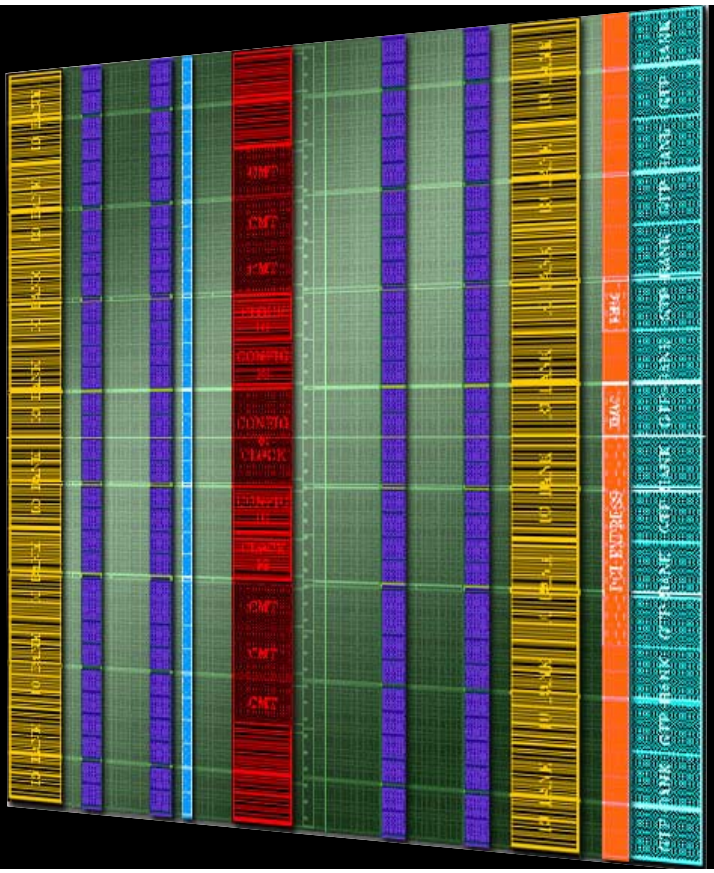
Example: given 1Kx8 SDP, want 1 read & 2 write ports.



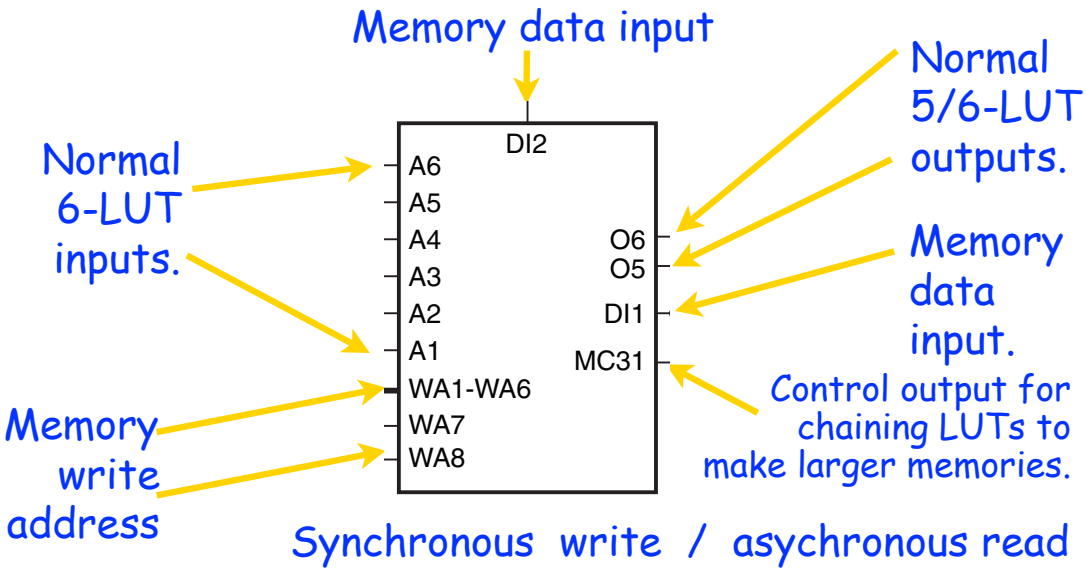
Virtex-5 LX110T memory blocks.

Distributed RAM using LUTs among the CLBs.

Block RAMs in four columns.



A SLICEM 6-LUT ...

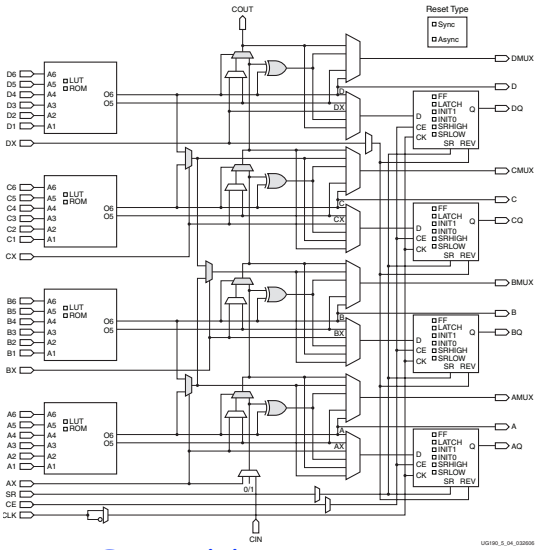


Synchronous write / asychronous read

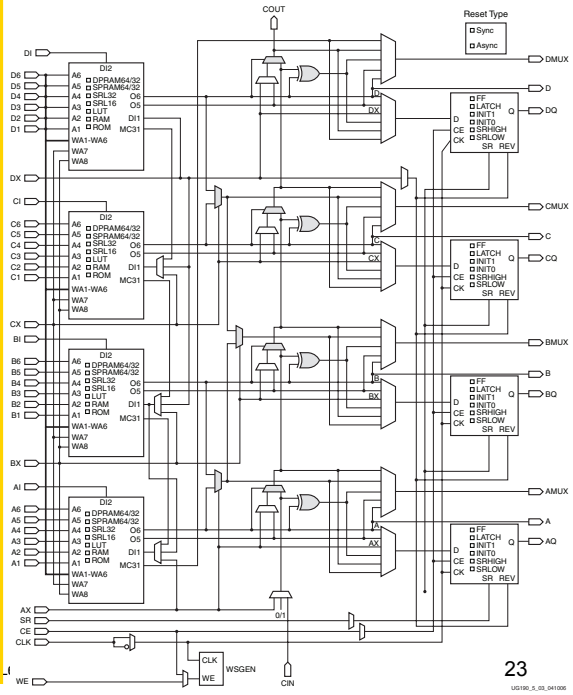
A 1.1 Mb distributed RAM can be made if all SLICEMs of an LX110T are used as RAM.

SLICEL vs SLICEM ...

SLICEL



SLICEM



SLICEM adds memory features to LUTs, + muxes.

Example Distributed RAM (LUT RAM)

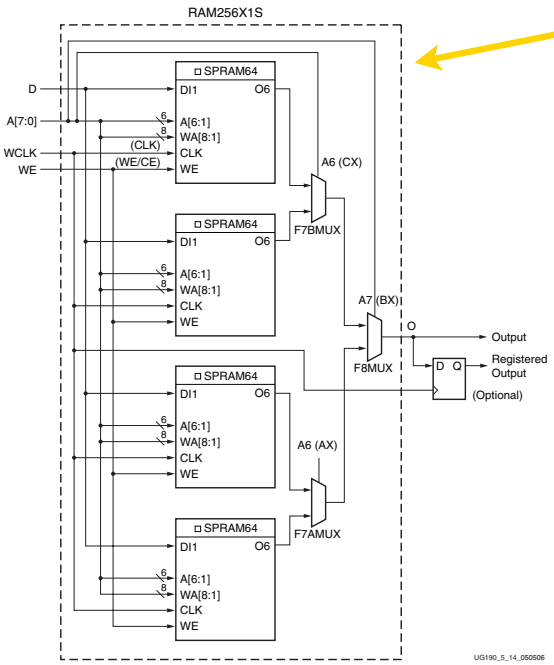
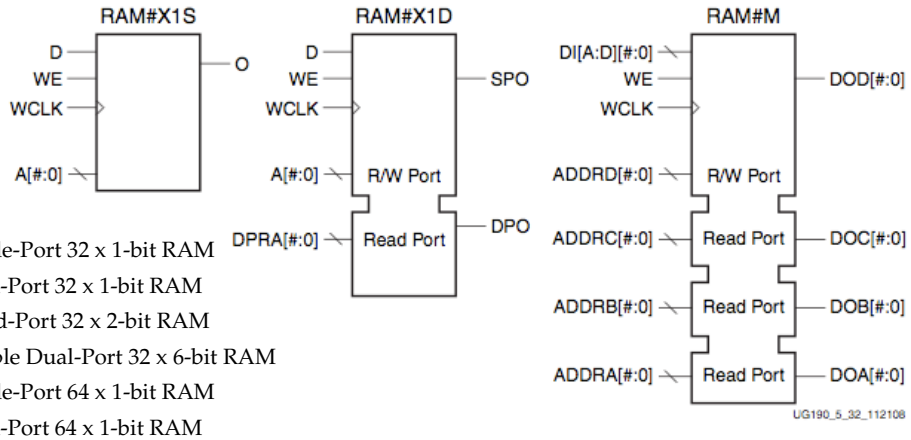


Figure 5-14: **Distributed RAM (RAM256X1S)**

Example configuration:
Single-port 256b x 1,
registered output.

A 128 x 32b LUT RAM has a 1.1ns access time.

Distributed RAM Primitives



- Single-Port 32 x 1-bit RAM
- Dual-Port 32 x 1-bit RAM
- Quad-Port 32 x 2-bit RAM
- Simple Dual-Port 32 x 6-bit RAM
- Single-Port 64 x 1-bit RAM
- Dual-Port 64 x 1-bit RAM
- Quad-Port 64 x 1-bit RAM
- Simple Dual-Port 64 x 3-bit RAM
- Single-Port 128 x 1-bit RAM
- Dual-Port 128 x 1-bit RAM
- Single-Port 256 x 1-bit RAM

All are built from a single slice or less.

Remember, though, that the SLICEM LUT is naturally only 1 read and 1 write port.

Example Dual Port Configurations

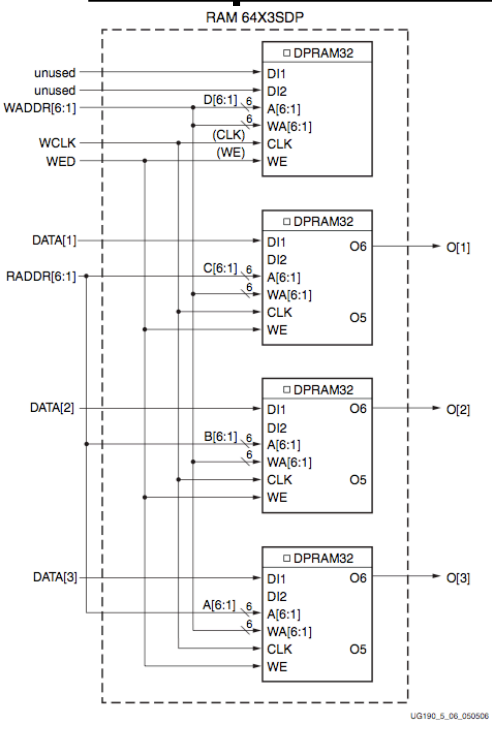


Figure 5-11: **Distributed RAM (RAM64X3SDP)**

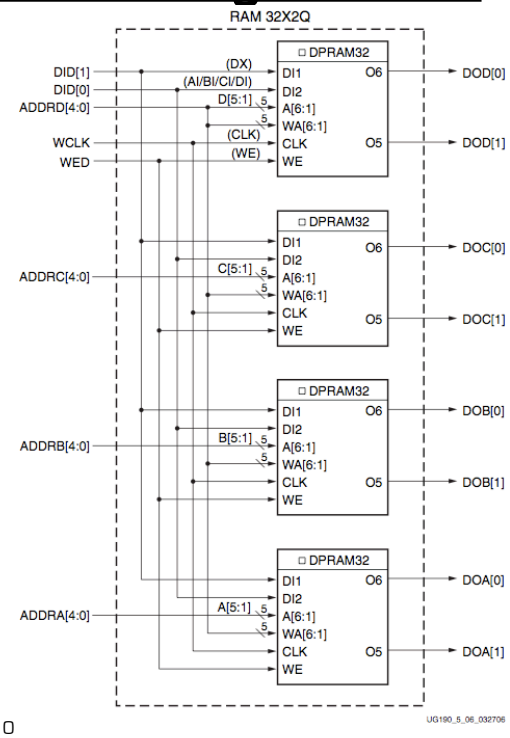


Figure 5-6: **Distributed RAM (RAM32X2Q)**

Distributed RAM Timing

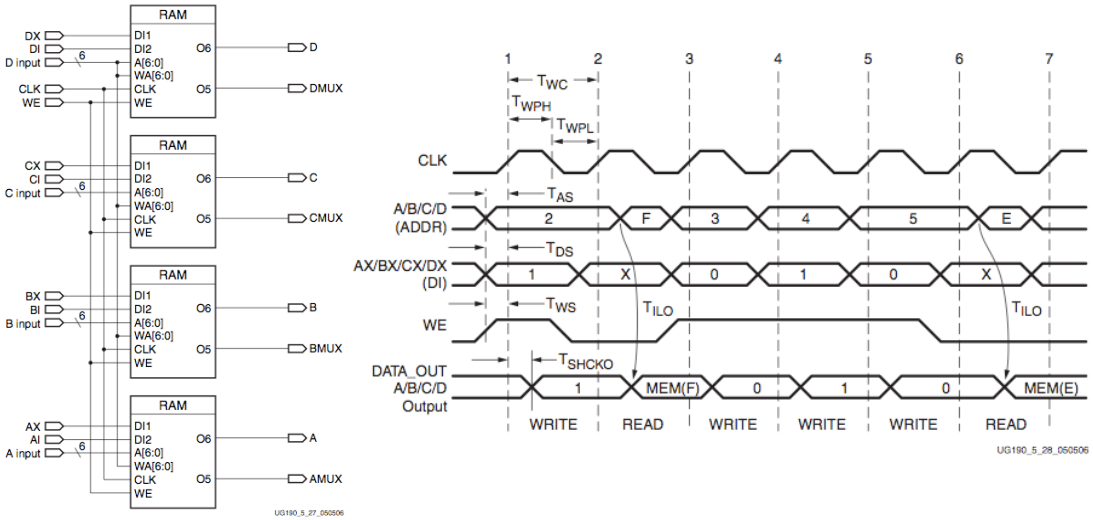
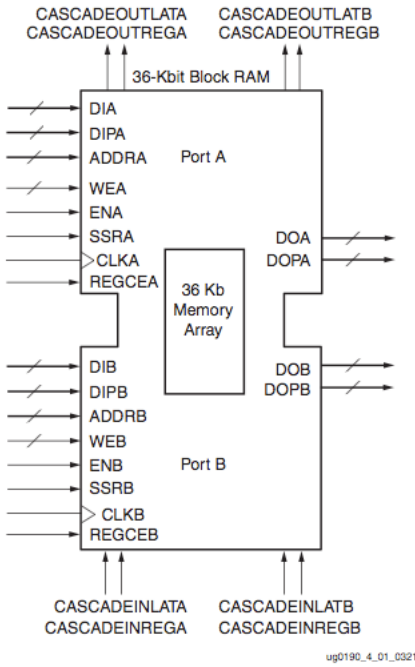


Figure 5-27: **Simplified Virtex-5 FPGA SLICEM Distributed RAM**

Table 1: **Virtex-5 FPGA Family Members**

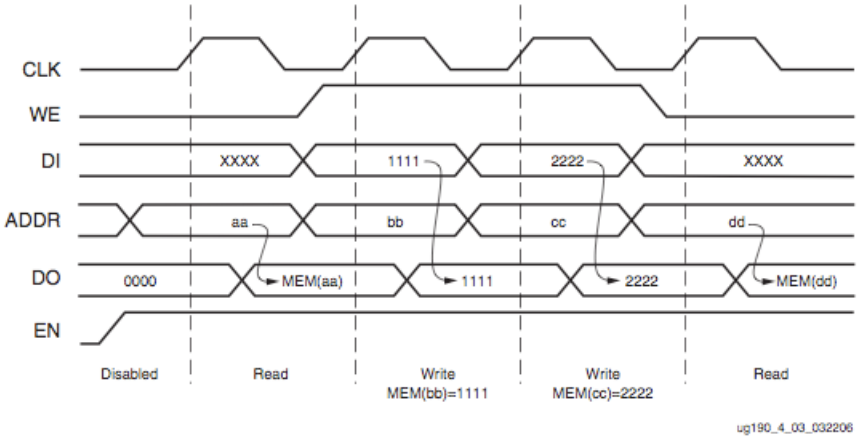
Device	Configurable Logic Blocks (CLBs)			DSP48E Slices[2]	Block RAM Blocks			CMTs[4]	PowerPC Processor Blocks	Endpoint Blocks for PCI Express	Ethernet MACs[5]	Max RocketIO Transceivers[6]		Total I/O Banks[8]	Max User I/O[7]
	Array (Row x Col)	Virtex-5 Slices[1]	Max Distributed RAM (Kb)		18 Kb[3]	36 Kb	Max (Kb)					GTP	GTX		
XC5VLX30	80 x 30	4,800	320	32	64	32	1,152	2	N/A	N/A	N/A	N/A	N/A	13	400
XC5VLX50	120 x 30	7,200	480	48	96	48	1,728	6	N/A	N/A	N/A	N/A	N/A	17	560
XC5VLX85	120 x 54	12,960	840	48	192	96	3,456	6	N/A	N/A	N/A	N/A	N/A	17	560
XC5VLX110	160 x 54	17,280	1,120	64	256	128	4,608	6	N/A	N/A	N/A	N/A	N/A	23	800
XC5VLX155	160 x 76	24,320	1,640	128	384	192	6,912	6	N/A	N/A	N/A	N/A	N/A	23	800
XC5VLX220	160 x 108	34,560	2,280	128	384	192	6,912	6	N/A	N/A	N/A	N/A	N/A	23	800
XC5VLX330	240 x 108	51,840	3,420	192	576	288	10,368	6	N/A	N/A	N/A	N/A	N/A	33	1,200
XC5VLX20T	60 x 26	3,120	210	24	52	26	936	1	N/A	1	2	4	N/A	7	172
XC5VLX30T	80 x 30	4,800	320	32	72	36	1,296	2	N/A	1	4	8	N/A	12	360
XC5VLX50T	120 x 30	7,200	480	48	120	60	2,160	6	N/A	1	4	12	N/A	15	480
XC5VLX85T	120 x 54	12,960	840	48	216	108	3,888	6	N/A	1	4	12	N/A	15	480
XC5VLX110T	160 x 54	17,280	1,120	64	296	148	5,328	6	N/A	1	4	16	N/A	20	680
XC5VLX155T	160 x 76	24,320	1,640	128	424	212	7,632	6	N/A	1	4	16	N/A	20	680
XC5VLX220T	160 x 108	34,560	2,280	128	424	212	7,632	6	N/A	1	4	16	N/A	20	680
XC5VLX330T	240 x 108	51,840	3,420	192	648	324	11,664	6	N/A	1	4	24	N/A	27	960
XC5VSX35T	80 x 34	5,440	520	192	168	84	3,024	2	N/A	1	4	8	N/A	12	360
XC5VSX50T	120 x 34	8,160	780	288	264	132	4,752	6	N/A	1	4	12	N/A	15	480
XC5VSX95T	160 x 46	14,720	1,520	640	488	244	8,784	6	N/A	1	4	16	N/A	19	640
XC5VSX240T	240 x 78	37,440	4,200	1,056	1,032	516	18,576	6	N/A	1	4	24	N/A	27	960
XC5VTX150T	200 x 58	23,200	1,500	80	456	228	8,208	6	N/A	1	4	N/A	40	20	680
XC5VTX240T	240 x 78	37,440	2,400	96	648	324	11,664	6	N/A	1	4	N/A	48	20	680
XC5VFX30T	80 x 38	5,120	380	64	136	68	2,448	2	1	1	4	N/A	8	12	360
XC5VFX70T	160 x 38	11,200	820	128	296	148	5,328	6	1	3	4	N/A	16	19	640
XC5VFX100T	160 x 56	16,000	1,240	256	456	228	8,208	6	2	3	4	N/A	16	20	680
XC5VFX130T	200 x 56	20,480	1,580	320	596	298	10,728	6	2	3	6	N/A	20	24	840
XC5VFX200T	240 x 68	30,720	2,280	384	912	456	16,416	6	2	4	8	N/A	24	27	960

Block RAM Overview



- 36K bits of data total, can be configured as:
  - 2 independent 18Kb RAMs, or one 36Kb RAM.
- Each 36Kb block RAM can be configured as:
  - 64Kx1 (when cascaded with an adjacent 36Kb block RAM), 32Kx1, 16Kx2, 8Kx4, 4Kx9, 2Kx18, or 1Kx36 memory.
- Each 18Kb block RAM can be configured as:
  - 16Kx1, 8Kx2, 4Kx4, 2Kx9, or 1Kx18 memory.
- Write and Read are synchronous operations.
- The two ports are symmetrical and totally independent (can have different clocks), sharing only the stored data.
- Each port can be configured in one of the available widths, independent of the other port. The read port width can be different from the write port width for each port.
- The memory content can be initialized or cleared by the configuration bitstream.

Block RAM Timing



- Note this is in the default mode, "WRITE_FIRST". Other possible modes are "READ_FIRST", and "NO_CHANGE".
- Optional output register, would delay appearance of output data by one cycle.
- Maximum clock rate, roughly 400MHz.

Verilog Synthesis Notes

- Block RAMS and LUT RAMS all exist as primitive library elements (similar to FDRSE). However, it is much more convenient to **use inference**.
- Depending on how you write your verilog, you will get either a collection of block RAMs, a collection of LUT RAMs, or a collection of flip-flops.
- The synthesizer uses size, and read style (synch versus asynch) to determine the best primitive type to use.
- It is possible to force mapping to a particular primitive by using synthesis directives. However, if you write your verilog correctly, you will not need to use directives.
- The synthesizer has limited capabilities (eg., it can combine primitives for more depth and width, but is limited on porting options). Be careful, as you might not get what you want.
- See **Synplify User Guide**, and **XST User Guide** for examples.

Inferring RAMs in Verilog

```
// 64X1 RAM implementation using distributed RAM

 module ram64X1 (clk, we, d, addr, q);
  input clk, we, d;
  input [5:0] addr;
  output q;

 reg [63:0] temp;
 always @ (posedge clk)
     if(we)
        temp[addr] <= d;
 assign q = temp[addr];

 endmodule
```

Verilog reg array used with "always @ (posedge ... infers memory array.

Asynchronous read infers LUT RAM

Dual-read-port LUT RAM

```
//
// Multiple-Port RAM Descriptions
//
module v_rams_17 (clk, we, wa, ra1, ra2, di, do1, do2);
    input  clk;
    input  we;
    input  [5:0] wa;
    input  [5:0] ra1;
    input  [5:0] ra2;
    input  [15:0] di;
    output [15:0] do1;
    output [15:0] do2;
    reg    [15:0] ram [63:0];
    always @(posedge clk)
    begin
        if (we)
            ram[wa] <= di;
    end
    assign do1 = ram[ra1];
    assign do2 = ram[ra2];
endmodule
```

Multiple reference to same array.

Block RAM Inference

```
//
// Single-Port RAM with Synchronous Read
//
module v_rams_07 (clk, we, a, di, do);
    input  clk;
    input  we;
    input  [5:0] a;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_a;
    always @(posedge clk) begin
        if (we)
            ram[a] <= di;
        read_a <= a;
    end
    assign do = ram[read_a];
endmodule
```

Synchronous read (registered read address) infers Block RAM

Block RAM initialization

```
module RAMB4_S4 (data_out, ADDR, data_in, CLK, WE);
   output[3:0] data_out;
   input [2:0] ADDR;
   input [3:0] data_in;
   input CLK, WE;
   reg [3:0] mem [7:0];
   reg [3:0] read_addr;

   initial
     begin
       $readmemb("data.dat", mem);
     end

   always@(posedge CLK)
     read_addr <= ADDR;

   assign data_out = mem[read_addr];

   always @(posedge CLK)
     if (WE) mem[ADDR] = data_in;

   endmodule
```

"data.dat" contains initial RAM contents, it gets put into the bitfile and loaded at configuration time. (Remake bits to change contents)

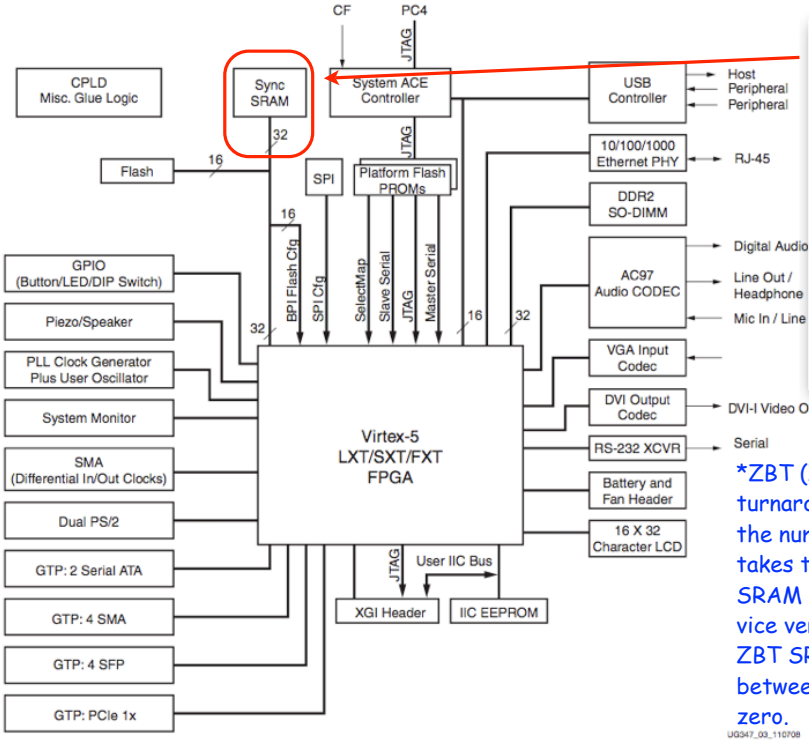
Dual-Port Block RAM

```
module test (data0,data1,waddr0,waddr1,we0,we1,clk0, clk1, q0, q1);

   parameter d_width = 8;  parameter addr_width = 8; parameter mem_depth = 256;

   input [d_width-1:0] data0, data1;
   input [addr_width-1:0] waddr0, waddr1;
   input we0, we1, clk0, clk1;

   reg [d_width-1:0] mem [mem_depth-1:0]
   reg [addr_width-1:0] reg_waddr0, reg_waddr1;
   output [d_width-1:0] q0, q1;

   assign q0 = mem[reg_waddr0];
   assign q1 = mem[reg_waddr1];

   always @(posedge clk0)
     begin
       if (we0)
         mem[waddr0] <= data0;
         reg_waddr0 <= waddr0;
     end

   always @(posedge clk1)
     begin
       if (we1)
         mem[waddr1] <= data1;
         reg_waddr1 <= waddr1;
     end

   endmodule
```

Processor Design Considerations (1/2)

- **Register File: Consider distributed RAM (LUT RAM)**
  - Size is close to what is needed: distributed RAM primitive configurations are 32 or 64 bits deep. Extra width is easily achieved by parallel arrangements.
  - LUT-RAM configurations offer multi-porting options - useful for register files.
  - Asynchronous read, might be useful by providing flexibility on where to put register read in the pipeline.
- **Instruction / Data Caches : Consider Block RAM**
  - Higher density, lower cost for large number of bits
  - A single 36kbit Block RAM implements 1K 32-bit words.
  - Configuration stream based initialization, permits a simple "boot strap" procedure.
- **Other Memories? FIFOs? Video "Frame Buffer"? How big?**

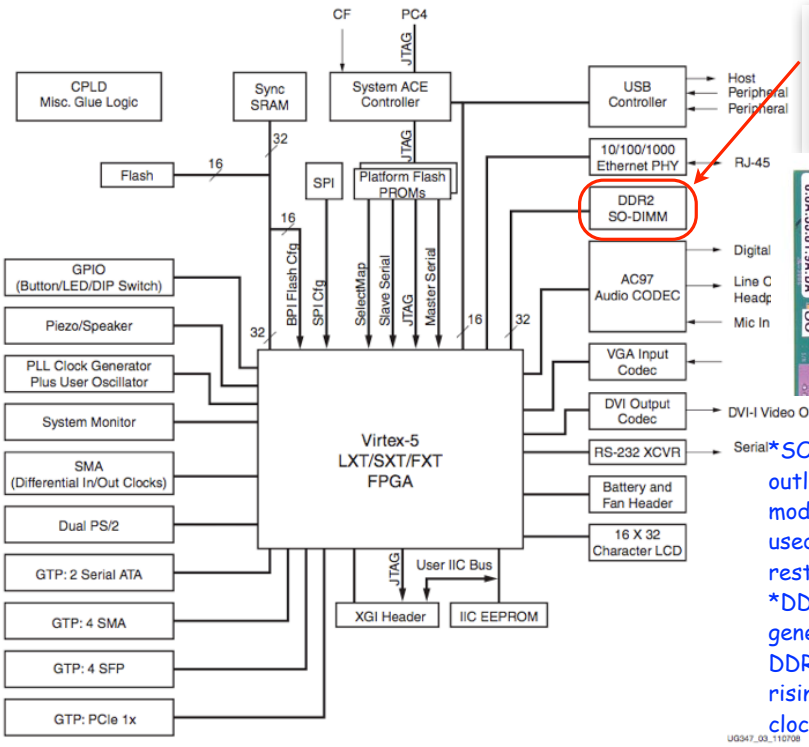
XUP Board External SRAM



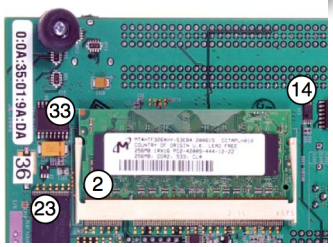
"ZBT" synchronous SRAM, 9 Mb on 32-bit data bus, with four "parity" bits
256K x 36 bits
(located under the removable LCD)

*ZBT (ZBT stands for zero bus turnaround) — the turnaround is the number of clock cycles it takes to change access to the SRAM from write to read and vice versa. The turnaround for ZBT SRAMs or the latency between read and write cycle is zero.

XUP Board External DRAM



256 MByte DDR2 DRAM with 400MHz data rate.



*SO-DIMM stands for small outline dual in-line memory module. SO-DIMMS are often used in systems which have space restrictions such as notebooks.

*DDR2 stands for second generation double data rate. DDR transfers data both on the rising and falling edges of the clock signal.