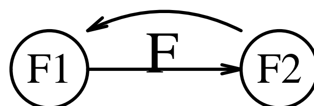
CMPT 250: Weeks 10-11 (Nov 7 to Nov 19)

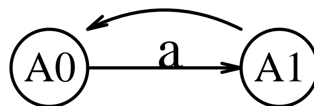
1. CENTRAL PROCESSING UNIT DESIGN (Continued)

2. CONTROLLER DESIGN FOR THE μMIPS MACHINE

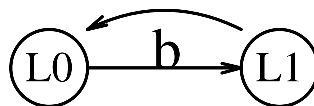
The design of the controller requires labelling the control inputs of the components in the datapath logic of the μMIPS machine. The control points indicated are from the diagram of the CPU at the end of last week's notes.

The design that evolves from an ASM specification results in a controller consisting of two components: a sequencer and a control-point enabler. Since there are five ASM diagrams, the sequencer can be specified by five state diagrams. The behavioral specification of the μMIPS sequencer consists of:

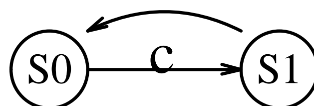




a = F' * op0



b = F' * op35



c = F' * op43

Because of the large number of control points in the datapath, a step-action sequence is used to describe the behavior of the control point enabler:

F1*F:	lir, cs, rm, spc,s1, lpc, d
F2:	la, lb, lalu, s1, kf
A0*F' *op0*fn32:	lalu, s1
A0*F' *op0*fn34:	lalu, s1, f_0
A0*F' *op0*fn36:	lalu, s1, f_2, f_0
A0*F' *op0*fn37:	lalu, s1, f_2, f_1
A0*F' *op0*fn42*altb:	sin1, sin0,swa, w, jf
A0*F' *op0*fn42*altb' :	sin1,swa, w, jf
A1:	w, swa, jf
L0*F' *op35:	lalu, s1, sop1
L1:	cs, rm, w, sin0, jf
S0*F' *op43:	lalu, s1, sop1
S1:	cs, jf
F' *op4*aeqb:	jf, lpc
F' *op4*aeqb' :	jf

Figure 2-1: Step-Action Sequence for the μMIPS Machine

The step-action sequence provides a suitable behavioral description for starting the design of the control point enabler. The overall design process is as follows:

1. Construct a word format sufficient to specify the values of all control points in the datapath, and any external status outputs that may also be required. This format is called a μ-instruction format.

2. Construct a μ-instruction for each step-action sequence control point list and store that μ-instruction in a word of ROM.

3. Define a mapping of the Boolean control functions for each step-action sequence to the address in ROM where the associated control point assignment is stored.

The mapping can be implemented using a programmable logic device such as a PLA (programmable logic array), to be described shortly.

Analysis of the DATAPATH for the CPU reveals that there are 14 status outputs corresponding to the opcode and fn-sel fields of the instruction register, the less-

than and equals outputs of the comparator, and the F flag. There are also 22 control points. Further the ASM diagram identifies 8 ASM blocks (not including the initialization state). The sequencer is implemented so that there is one output/state and therefore will supply a further 8 inputs to the control point enabler. Thus the control point enabler potentially has 22 inputs and 22 outputs.

3. PROGRAMMABLE LOGIC DEVICES (PLDs)

In contemporary logic design, combinational circuits are created from more complex components rather than from discrete logic gates. However minimization techniques associated with discrete gate implementations are still important in many instances, including the implemention of combinational logic using programmable logic devices (PLDs) which include the following:

- ROMs: Read only memory
- PALs: Programmable array logic
- PLAs: Programmable logic arrays

Implementing combinational logic using programmable logic devices produces implementations that are smaller, cheaper, and easier to fix or modify than conventional gate-level implementations.

Programmable logic devices provide a matrix of potential connections for implementing a set of Boolean functions expressed as sums-of-products. Initially fusible connections are provided at all intersections of lines in the schematic of the organization of a hypothetical PLA on the next page.

The implementation of a combinational circuit with a PLD can be expressed in a `PLD schematic`. This diagram specifies (with an "X") the connections to preserve in a "simplifed" logic diagram of a PLD. (see next page). This diagram represents a set of combinational circuits, each expressed as a sum of products. Each horizontal line defines a product term. Each vertical line to the left of the AND gates defines a different literal from the inputs. Each vertical line to the right of the AND gates defins a sum of products.

The intersections of horizontal and vertical lines to the left of the AND gates is called the *product array* while those to the right define the *sum array*. "X"'s on any horizontal line in the product array define the literals that are ANDed together by the AND gate on that horizontal line. "X"'s on any vertical line of the sum array define the product terms that are summed in the OR gate at the bottom of the vertical line which determines that output as a sum of products.

The PLD is "programmed" by specifying which intersecting connections are to be disconnected. A device, called a "logic programmer", performs this task from a tabular specification of the the locations where the fusible connections should be broken.

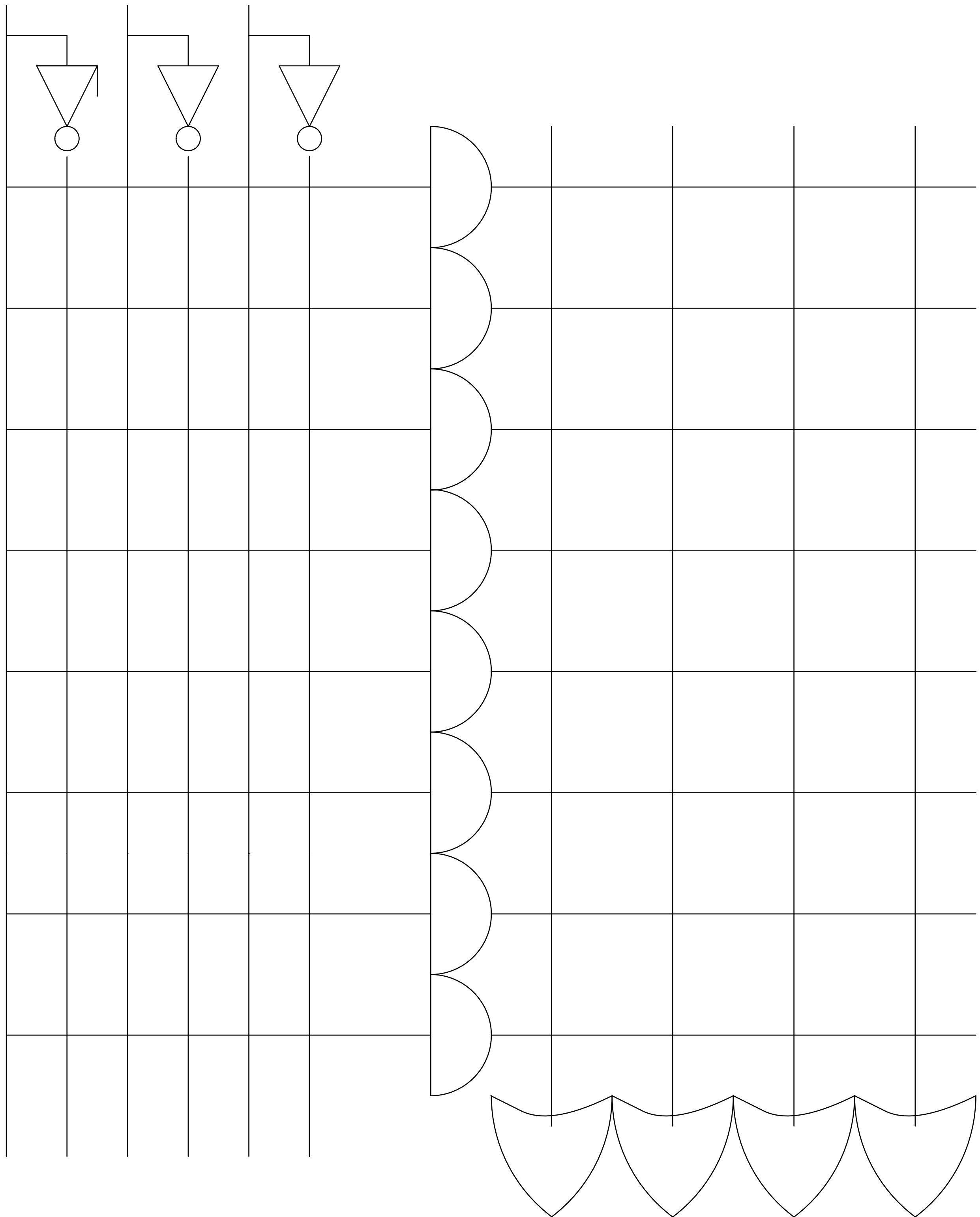


Figure 3-1: Schematic of a 3x4 PLD with 8 product terms

A PAL is a PLD where only the product terms can be programmed. Each sum is already to "hardwired" to a fixed number of products. Thus every function to be implemented with a PAL must be expressible using no more poroduct terms than the PAL provides per output.

A ROM is a PLD where only the sum terms can be programmed. In a ROM every minterm that can be defined on the inputs is provided as a product term. Each minterm actually corresponds to a location in memory expressed as a sequence of values assigned to the variables. Then the collective values of all functions for a particular assignment of values to the variables defines the contents of the word of memory at that location. In effect the ROM stores the function table of the function being defined.

A PLA permits the programming of both the product-terms and sum-terms.

On the next page is illustrated an example implementing the following step-action sequence using a PLA. The structural schematic and an alternative way of providing a behavioral descripton, called a *personality matrix*, are provided. The initial step-action sequence is:

```
x*y      : f2, f0
x'*y'*z': f2
z        : f1, f0
y'*z     : f2
```

From this step-action sequence, one obtains the following sums-of-products:

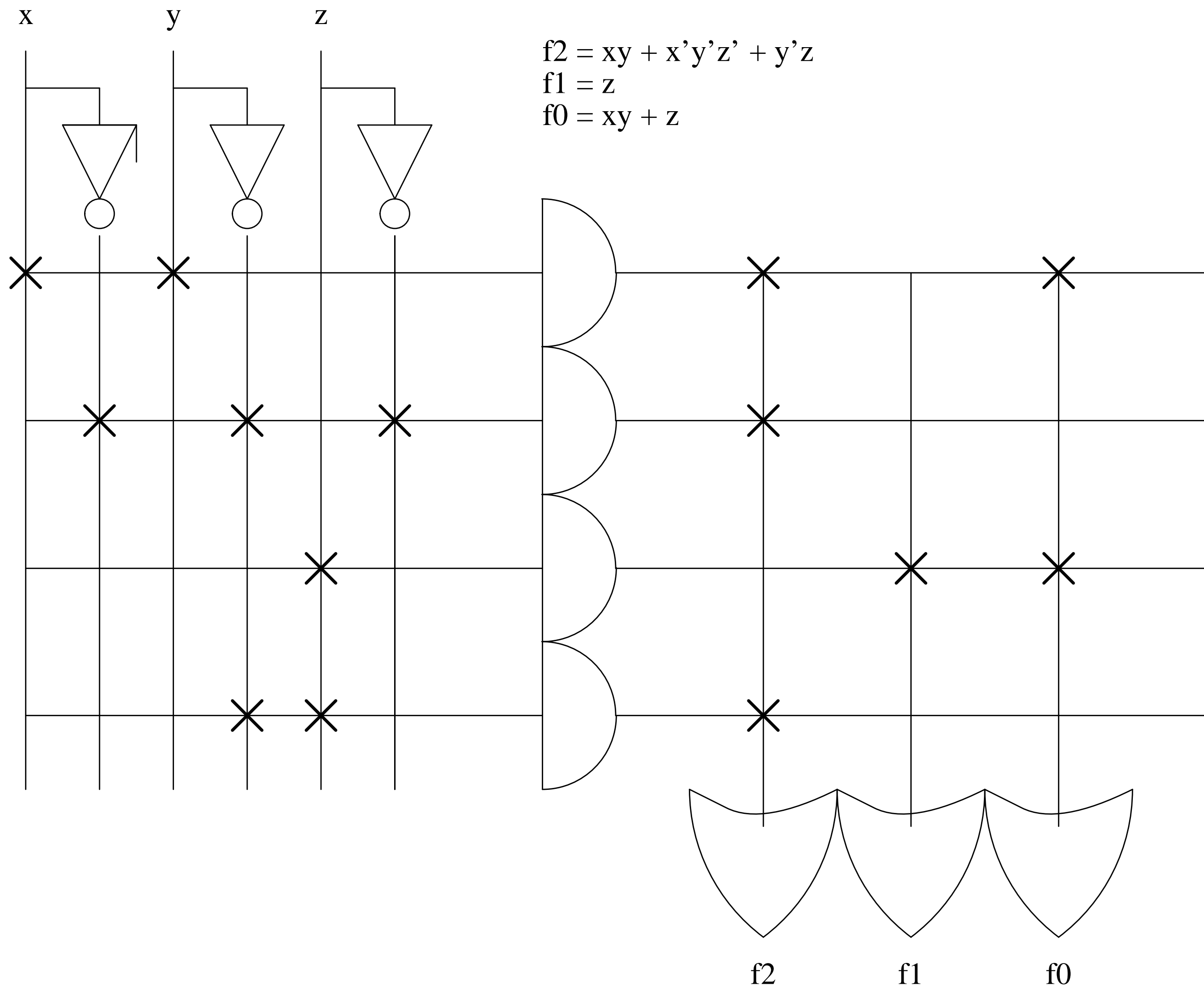
f0 = x*y + z

f1 = z

f2 = x*y + x'*y'*z' + y'z

There are four distinct product terms required as suggested by the existence of four step-action statements whose boolean control expressions are product terms.

From this information we can identify the connections that are to be retained on a PLA scehmatic (next page):



PRODUCT TERM	INPUTS x y z	OUTPUTS f2	f1	f0
xy	11-	1	-	1
x'y'z'	000	1	-	-
z	--1	-	1	1
y'z	-01	1	-	-

Figure 3-2: PLD and associated personality matrix

By representing control words as encoded sequences, the control point enabler can be implemented using a ROM to store the μinstructions for each control point sequence, and a PLA to select the correct μinstruction from the ROM.

4. MEMORY ARCHITECTURES

Memory can be provided using any or all of five different types of "standard" components:

- **Flip-flops**: Each component provides 1 bit of storage. Each device includes a data input a data output and control points to permit storage of the input. A tri-state buffer can be added to control accessibility to the output. Flip-flops by definition include a clock enable input and are therefore synchronous devices
- **Registers**: Each component provides a word of storage. In addition to a parallel input and output permitting simultaneous access to all bits of the word, additional control points permit storage and possibly some operational capability (such as incrementing or shifting). Tri-state logic may be included to control accessibility to the output. Registers include a clock input and are therefore synchronous.
- **Register Files**: Each component provides a set of addressable registers and set of address inputs to select the registers to accessed for input and output. Registers may be retrieved from and stored to simultaneously, depending on the status of read and write control points. A clock input is also available, permitting register files to be used as synchronous components.
- **Static RAM (SRAM)**: Each component provides a set of addressable words that can be either written to or retrieved from, but not both at the same time. The content of memory remains unchanged when not enabled for read or write by a `cs` control point, as long as power is supplied. The access function is determined by a `r/w` function select control point. SRAM uses a D-latch as the basis for implementing each bit of storage, and is therefore an asynchronous device. The data bus is typically bidirectional.
- **Dynamic RAM (DRAM)**: Similar to SRAM, but the bits of memory are implemented using a capacitor, with the stored value represented by the charge on the capacitor. Because the charge decays with time, the device is called *dynamic* since the value stored does not remain indefinitely without exteranl assistance. In addition to the control points of an SRAM, a DRAM also has a *column strobe* and *row strobe*

to permit the address to be provided in two parts and to provide internal timing.

SRAM and DRAM are asynchronous devices. To interface these with a synchronous design, it is necessary to provide a controller and possibly additional logic to accommodate the timing characteristices of the SRAM or DRAM.

In a memory device (synchronous or asynchronous), the *memory cycle time* is the minimum time between two successive memory accesses. This time can be either a *read-cycle time* or a *write-cycle time* depending on whether the accesses are for retrieval or storage.

A synchronous memory architecture can be obtained using an SRAM or DRAM and then introducing registers and a clocked memory controller. This architecture must then generate an `ack` signal (acknowledge) to indicate when the memory has completed the task requested. This permits the memory to have a different clock period than the CPU, one sufficient to accommodate the set-up and hold times and propagation delay of the asynchronous memory matrix.

The following ASM captures the related activity in a CPU and a MEMORY with a synchronous controller.

The memory cycle time is the principal bottleneck in a Von Neumann computer architecture, since every instruction and data item on which an instruction may be performed is stored in the memory unit.

The technology adopted for implementing memory is the principal factor affecting the memory cycle time. Static RAM (SRAM) is based on implementing each bit of storage using the equivalent of a latch. For larger memories a simpler design is required for storing a bit. This is achieved in Dynamic RAM.

Each DRAM cell consists of a small capacitor and a transistor that acts like a switch. When the cell is selected the switch permits the capacitor to be charged, representing the storage of a '1' bit, or discharged, representing the storage of a '0' bit. The value stored is defined by the voltage level on the data line.

To retrieve a value, it is necessary to determine if the capacitor is charged. A voltage level half-way between logic-1 and logic-0 is placed on the data line, and the select line is enabled. The result is that the voltage level increases slightly if the capacitor is already charged, and decreases slightly if the capacitor is not already charged. By sensing (with amplifiers) this increase or decrease in voltage on the data line, it is possible to interpret the value that was stored there.

In designing a memory based on DRAM technology it is therefore important to address two issues:

- Retrieval destroys memory contents.
- Capacitors lose their charge with time.

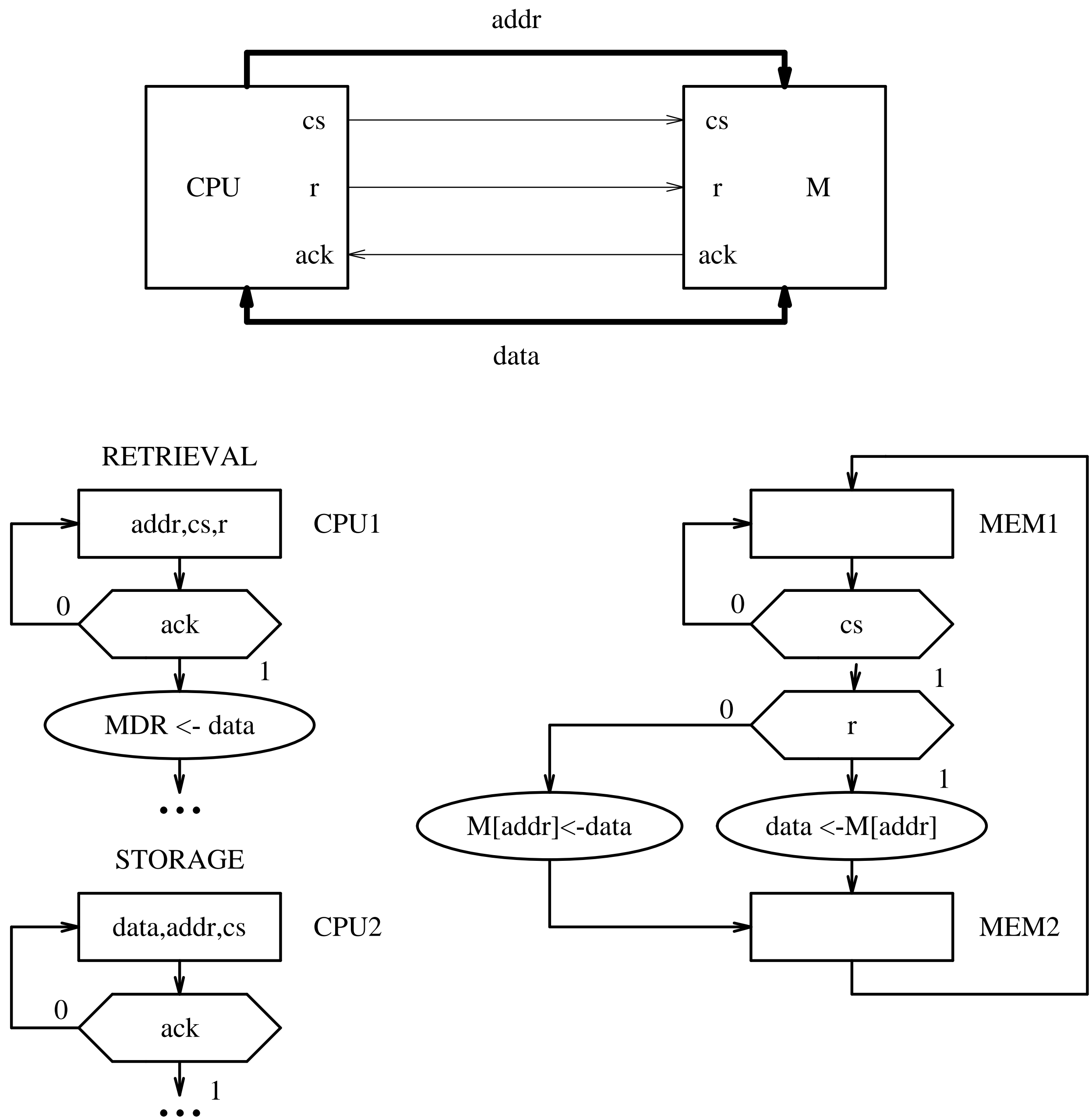
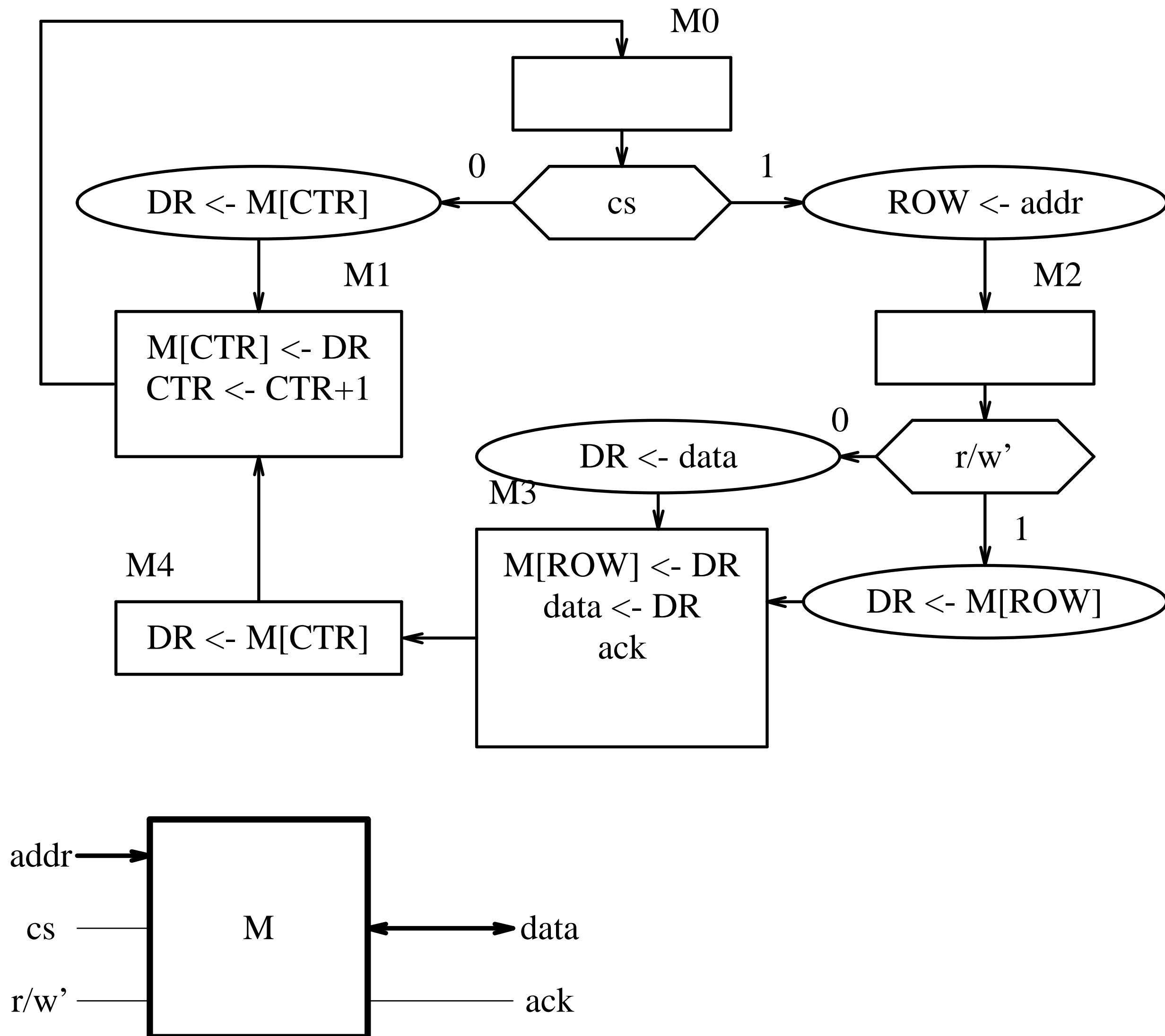


Figure 4-1: CPU/Memory interface

So it is necessary to refresh memory whenever a value is retrieved and in any event to refresh it periodically, to avoid loss from capacitor decay, typically from 2ms to 4ms upto as long as 64 ms in newer designs.

5. DRAM MEMORY ARCHITECTURE

To perform a memory access request, a dynamic memory requires a sequence of steps to be performed to complete an access. The following ASM diagram addresses the special requirements of dynamic memory. State box M1 uses a counter to sequentially access locations for refreshing. Each location is read into a buffer register DR whose contents are then copied back to their original location.



Whenever a memory request is initiated, the value being read or written is copied first into the buffer register, DR. This permits the value then to be written to memory, either as a new value (if a write request was made) or as a restore of the original value that was lost because of the destructive aspect of a read. Note than even when a memory request is processed, at least one additional refresh occurs before a check is made for a new request. This ensures that the refresh "loop" is not locked out if a large succession of memory requests is made.

A difficulty that arises with the need to refresh memory as suggested by the ASM diagram is that only so many locations can be refreshed before the locations not yet refreshed begin to decay. The number of locations depends on the time spent to refresh each location.

A logical way to increase the maximum number of locations that can be maintained is to allow more than one location to be refreshed at a time. This requires a larger buffer register (DR) capable of holding a "row" of memory words, and select logic to access the desired word within the row.