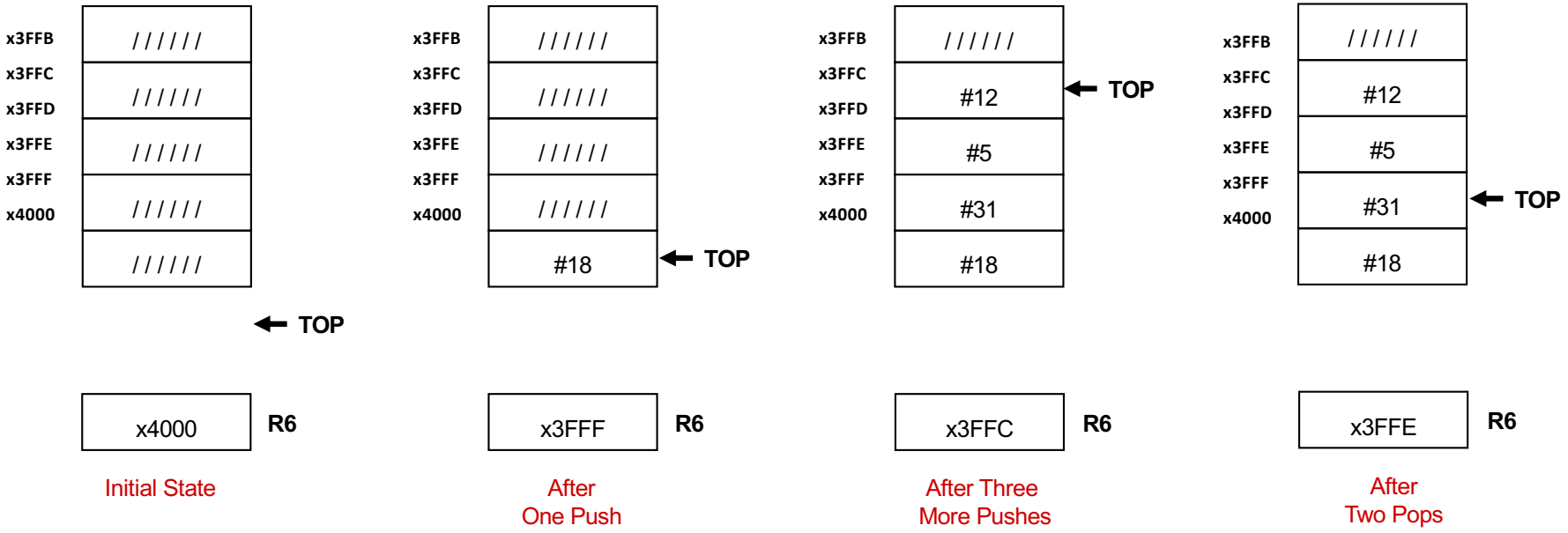
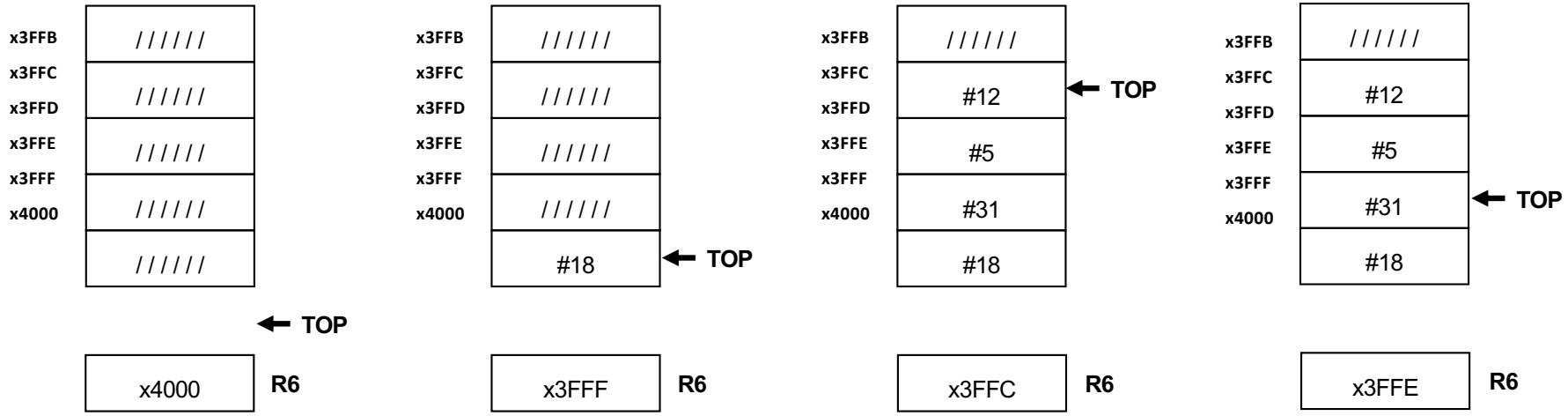
A Software Stack Implementation

- Data items don't move in memory, just our idea about where TOP of the stack is



By convention, R6 holds the Top of Stack (TOS) Pointer (SP)

Basic Push and Pop Code




```
PUSH
      ADD R6, R6, #-1                  ; increment stack ptr
      STR R0, R6, #0           ; store data(R0) to TOS
POP
      LDR R0, R6, #0           ; load data(R0) from TOS
      ADD R6, R6, #1           ; decrement stack ptr
```

- Note: Stacks can grow in either direction (toward higher address or toward lower addresses)

Pop with Underflow Detection

- If we try to pop too many items off the stack, an underflow condition occurs.
  - Check for underflow by checking TOS before removing data.
  - Return status code in R5 (0 for success, 1 for underflow)

```
POP   LD  R1, EMPTY
      ADD R2, R6, R1          ; Compare stack pointer
      BRz UNDER               ; with x3FFF
      LDR R0, R6, #0       ; The actual 'pop'
      ADD R6, R6, #1          ; Adjust stack pointer
      AND R5, R5, #0          ; Success: return R5 = 0
      RET
UNDER AND R5, R5, #0          ; Underflow: return R5 = 1
      ADD R5, R5, #1
      RET
EMPTY .FILL xC000             ; EMPTY = -x4000
```

Push with Overflow Detection

- If we try to push too many items onto the stack, an overflow condition occurs.
  - Check for underflow by checking TOS before adding data.
  - Return status code in R5 (0 for success, 1 for overflow)

```
PUSH  LD  R1, FULL
      ADD R2, R6, R1          ; Compare stack pointer
      BRz OVER                ; with x4004
      ADD R6, R6, #-1         ; Adjust stack pointer
      STR R0, R6, #0          ; The actual 'push'
      AND R5, R5, #0          ; Success: return R5 = 0
      RET
OVER  AND R5, R5, #0
      ADD R5, R5, #1; Overflow: return R5 = 1
      RET
FULL  .FILL xC005             ; FULL = -x3FFB
```

14.15 The following C program is compiled into LC-3 machine language and loaded into address x3000 before execution. Not counting the JSRs to library routines for I/O, the object code contains three JSRs (one to function `f`, one to `g`, and one to `h`). Suppose the addresses of the three JSR instructions are x3102, x3301, and x3304. Also suppose the user provides `4 5 6` as input values. Draw a picture of the run-time stack, providing the contents of locations, if possible, when the program is about to return from function `f`. Assume the base of the run-time stack is location xEFFF.

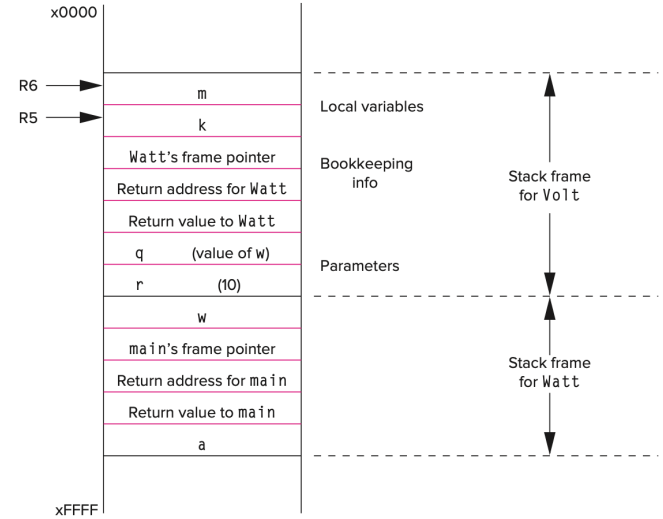


Figure 14.7 The run-time stack after the stack frame for `Volt` is pushed onto the stack.

```
#include <stdio.h>
int f(int x, int y, int z);
int g(int arg);
int h(int arg1, int arg2);

int main(void)
{
   int a, b, c;

   printf("Type three numbers: ");
   scanf("%d %d %d", &a, &b, &c);
   printf("%d", f(a, b, c));
}

int f(int x, int y, int z)
{
   int x1;

   x1 = g(x);
   return h(y, z) * x1;
}

int g(int arg)
{
   return arg * arg;
}

int h(int arg1, int arg2)
{
   return arg1 / arg2;
}
```

2.1 The Definition of Queue

- Queue
  - A data structure with the property of “First in First out”;
  - Front pointer for removing elements from the front of the queue; Rear pointer for inserting into the rear of the queue.
  - Front points to the location just in front of the first element in the queue; Rear points to the location of the last element in the queue.

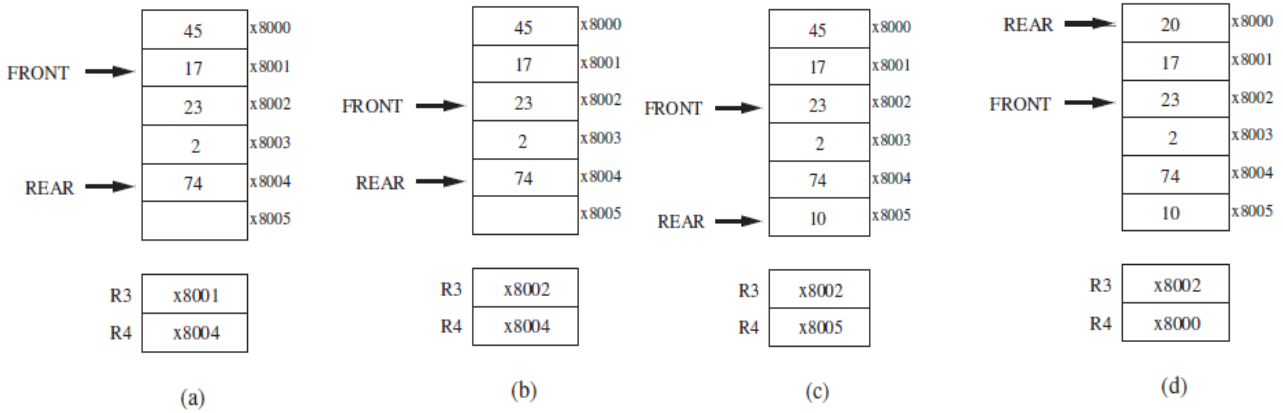
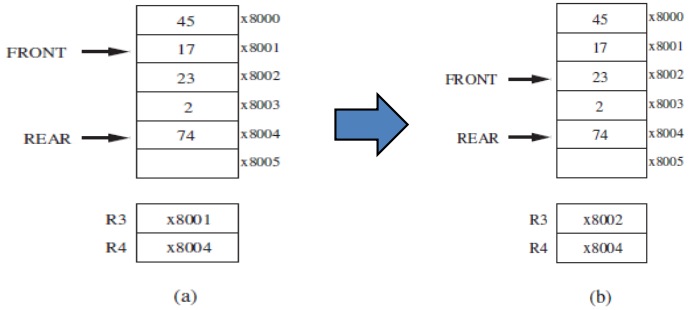


Figure 8.25 A queue allocated to memory locations x8000 to x8005.

2.2 Basic Operations

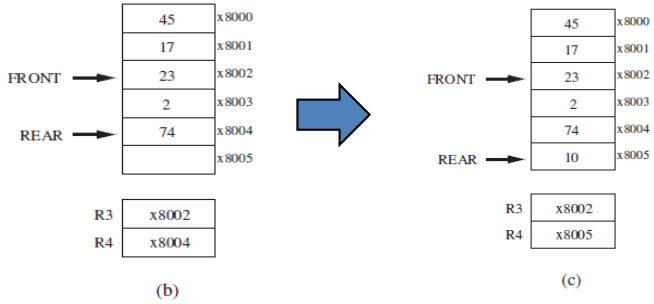
- Remove from Front
  - FRONT points to the location just in front of the first element in the queue; R3 stores the FRONT pointer;
  - First incrementing FRONT; then loading the value.

```
ADD R3,R3,#1
LDR  R0,R3,#0
```



- Insert at Rear
  - First incrementing REAR; then storing the value. R4 stores the REAR pointer;

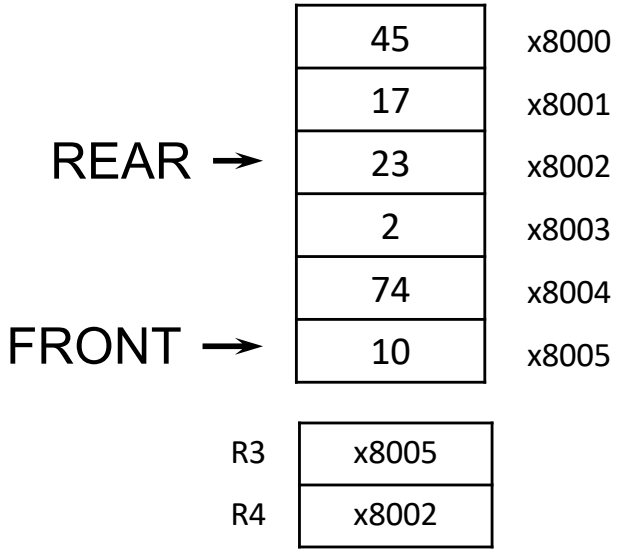
```
ADD R4,R4,#1
STR  R0,R4,#0
```



2.3 Wrap-Around

- Remove

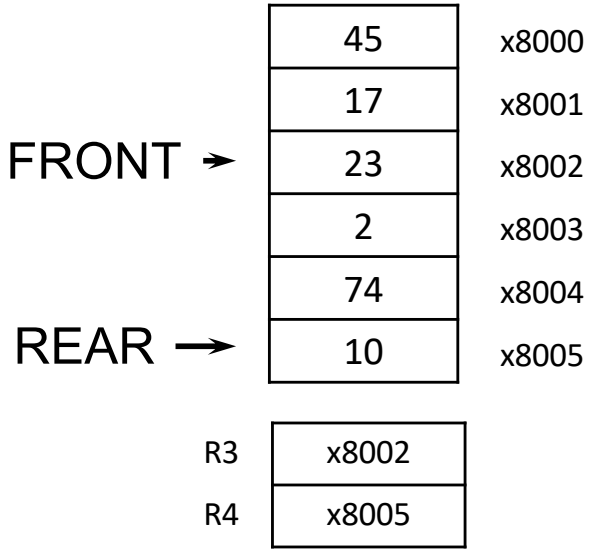
```
        ;R3 stores FRONT pointer
        LD R2, LAST
        ADD R2,R3,R2
        BRnp SKIP_1
        LD R3,FIRST
        BR SKIP_2
SKIP_1 ADD R3,R3,#1
SKIP_2 LDR  R0,R3,#0
; R0 gets the front of the queue
        RET
LAST    .FILL  x7FFB
; LAST contains the negative of 8005
FIRST   .FILL  x8000
```



2.3 Wrap-Around

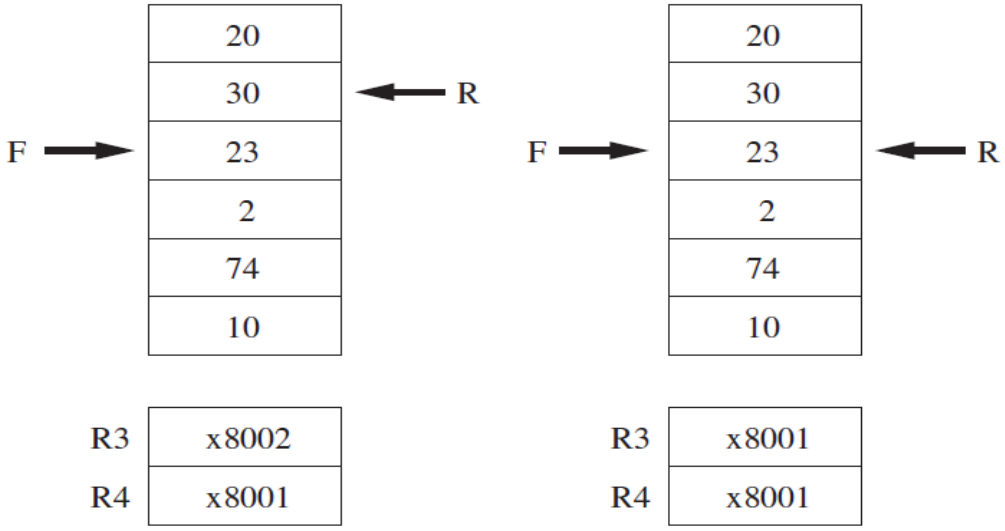
■ **Insert**

```
        ;R4 stores REAR pointer
        LD R2, LAST
        ADD R2,R4,R2
        BRnp SKIP_1
        LD R4,FIRST
        BR SKIP_2
SKIP_1 ADD R4,R4,#1
SKIP_2 STR R0,R4,#0
; R0 gets the front of the queue
        RET
LAST    .FILL 7FFB
; LAST contains the negative of 8005
FIRST   .FILL x8000
```



2.4 Full and Empty Queue

- The queue are allowed to store only n-1 elements for a queue with n locations.
- Full: FRONT=REAR+1 mod n Empty: FRONT=REAR

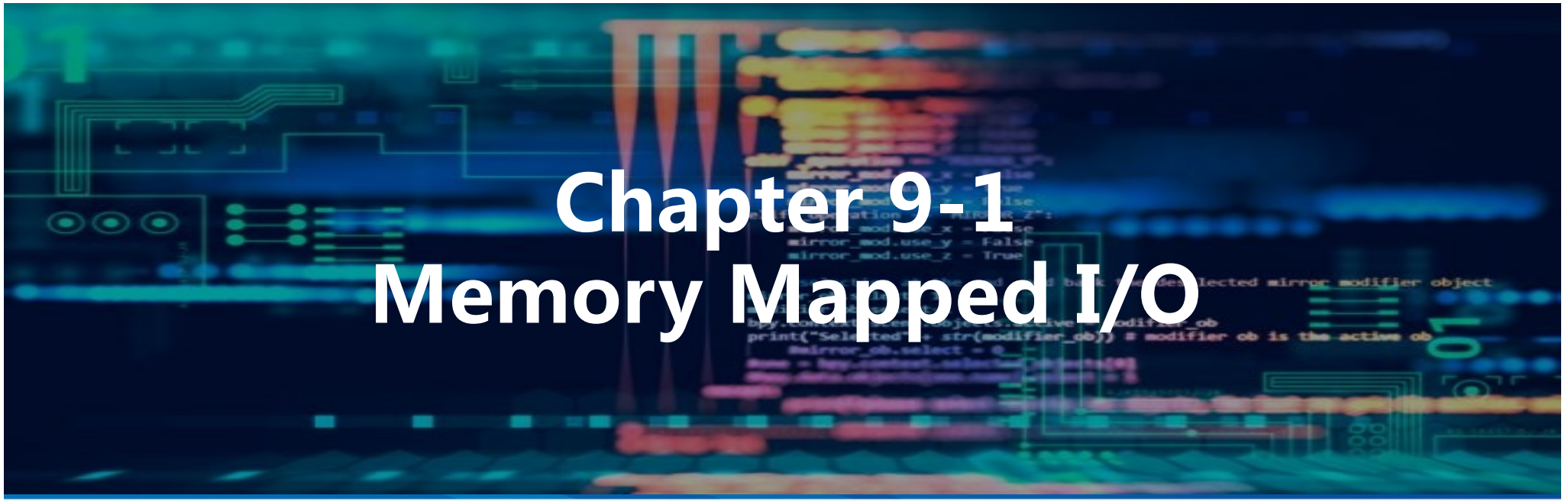


(a) A full queue (b) An empty queue

Figure 8.26 A full queue and an empty queue.

中国科学技术大学
University of Science and Technology of China

计算系统概论A
Introduction to Computing Systems
（ CS1002A.03 ）



Chapter 9-1
Memory Mapped I/O

陈俊仕
cjuns@ustc.edu.cn
2023 Fall

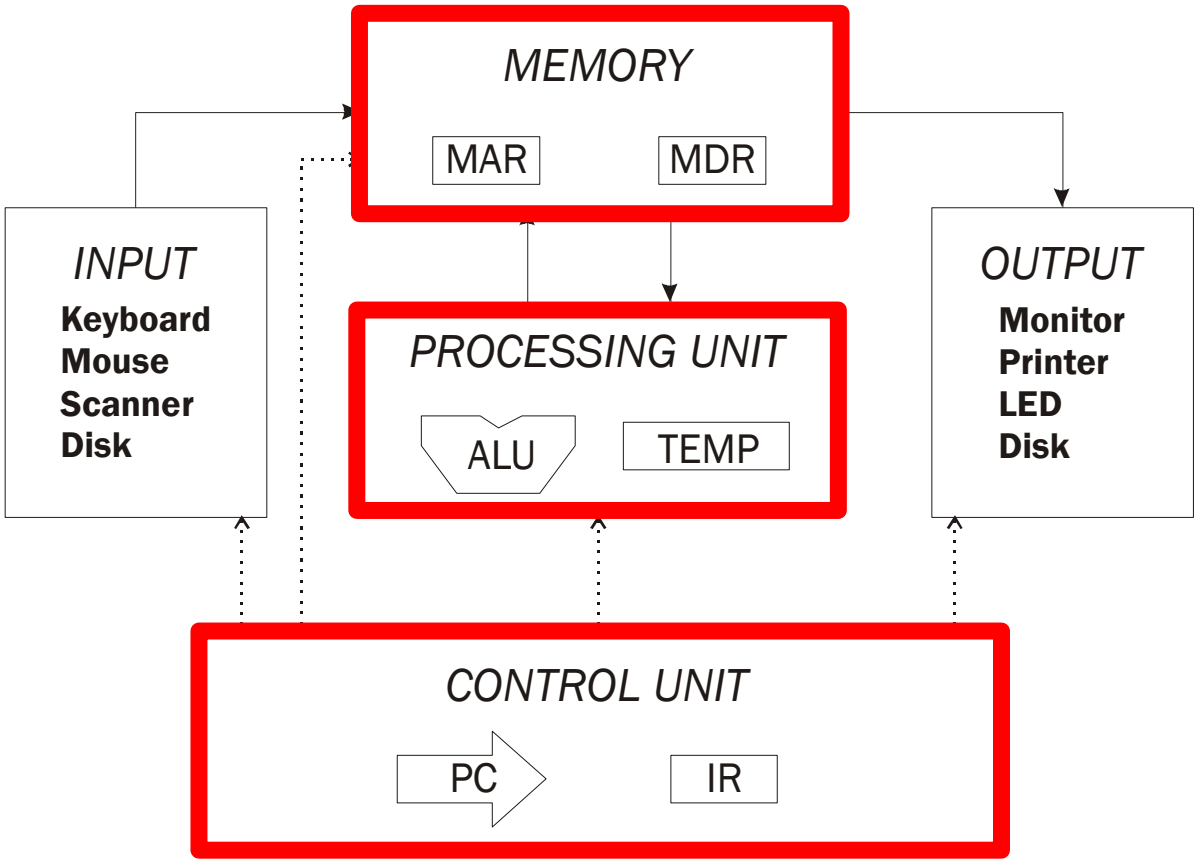
计算机科学与技术学院
School of Computer Science and Technology

Outline

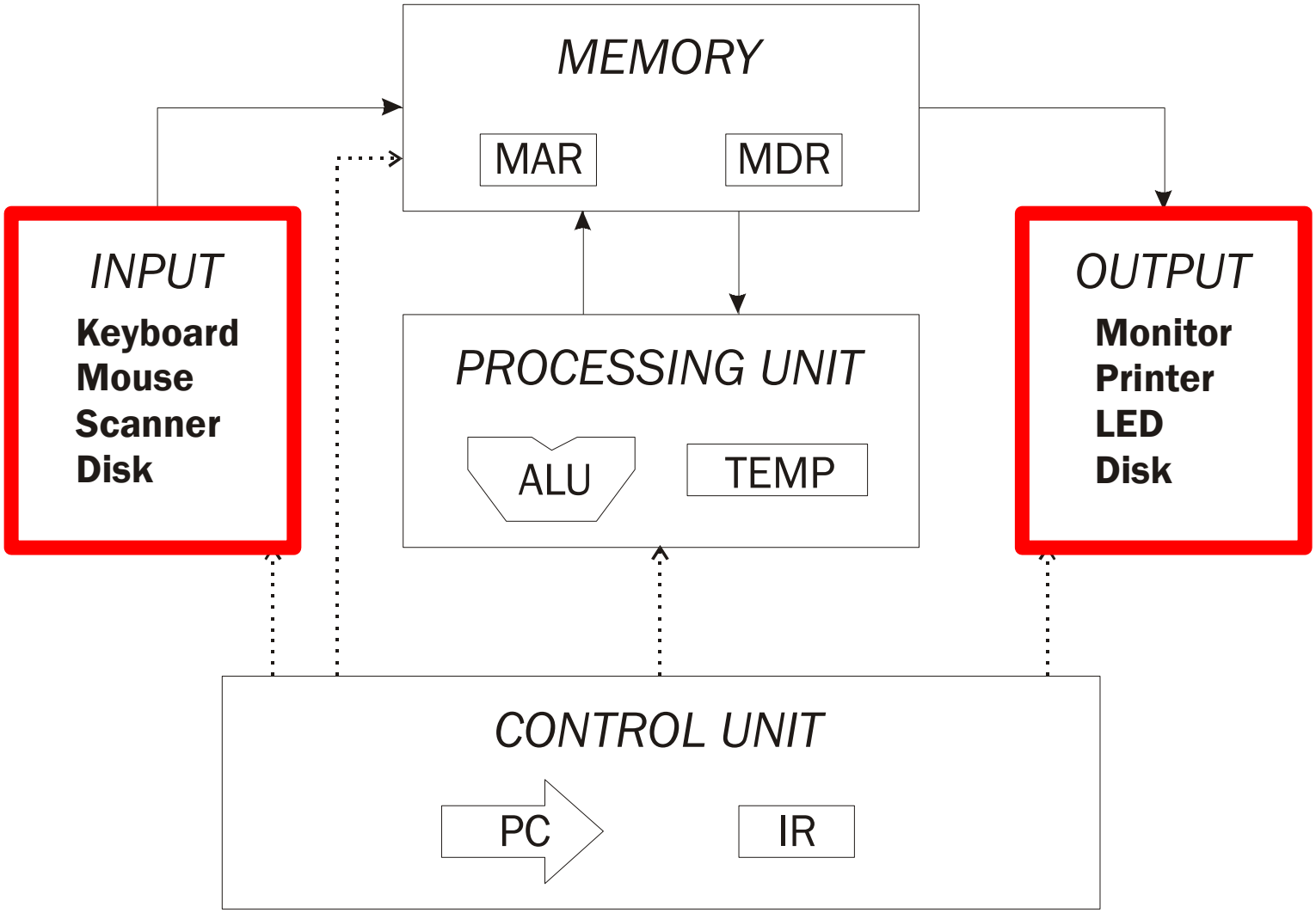
中国科学技术大学
University of Science and Technology of China

Review

- **So far, we've learned how to:**
  - compute with values in registers
  - load data from memory to registers
  - store data from registers to memory



Today: I/O in Von Neumann Model



Outline

中国科学技术大学
University of Science and Technology of China

Need for I/O

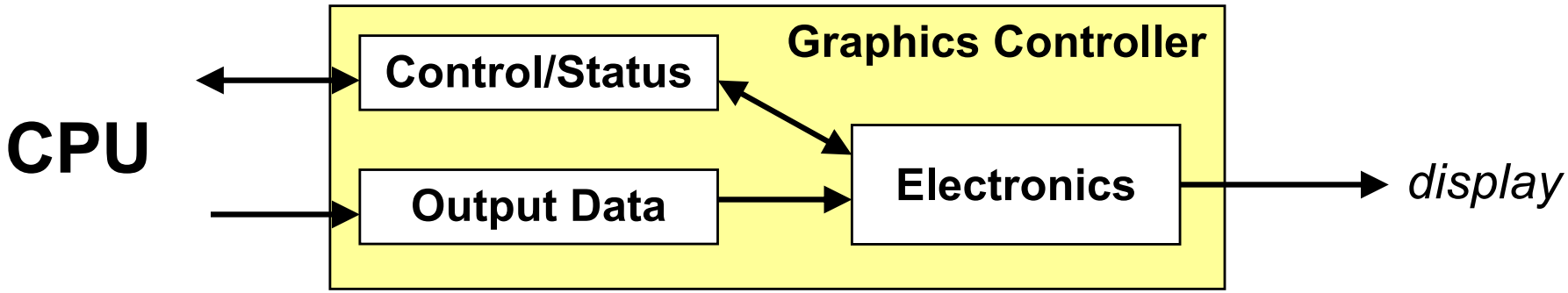
- But where does data in memory come from? And how does data get out of the system so that humans can use it?
  - Computer systems are useless unless they can process information from outside of the computer and output results outside of the computer
  - I/O is effectively the communication with outside of the computer
- I/O device itself communicates with outside world
  - E.g., keyboard takes input from user
- Computer needs to communicate with I/O device
  - E.g., computer takes input from keyboard
- Communication through shared memory locations
  - Processor and I/O can read/write those memory locations
  - Sometimes, data in memory locations can be set/cleared automatically (by hardware) depending on a read/write

I/O: Connecting to the Outside World

- **Examples**
  - **Keyboard/mouse input, video output on a standard computer**
  - **Network input/output that enables web surfing**
  - **Information from an engine of a car that a computer uses to determine how to tune the engine (output from computer tunes the engine)**
  - **Requests for airline reservations and replies that service those requests**
- **Types of I/O devices characterized by:**
  - **behavior: input, output, storage**
    - **input: keyboard, motion detector, network interface**
    - **output: display screen(monitor), printer, network interface**
    - **storage: disk, CD-ROM**
  - **data rate: how fast can data be transferred?**
    - **keyboard: 100 bytes/sec**
    - **disk: 60-120 MB/s**
    - **network: 1 Mb/s - 100 Gb/s**

I/O Controller

- **Control/Status Registers**
  - CPU tells device what to do -- write to control register
  - CPU checks whether task is done -- read status register
- **Data Registers**
  - CPU transfers data to/from device



- **Device electronics**
  - performs actual operation
    - pixels to screen, bits to/from disk, characters from keyboard

Programming Interface

- How are device registers identified?
  - Memory-mapped vs. special instructions
- How is timing of transfer managed?
  - Asynchronous vs. synchronous
- Who controls transfer?
  - CPU (polling) vs. device (interrupts)

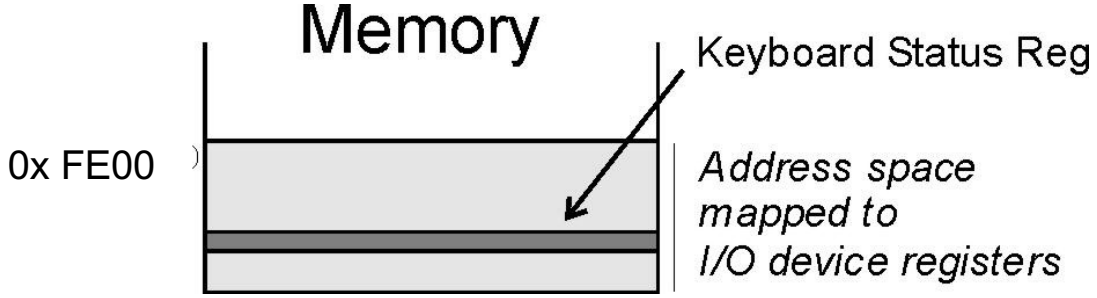
Memory-Mapped vs. I/O Instructions

Instructions

- designate opcode(s) for I/O
- register and operation encoded in instruction

Memory-mapped

- assign a memory address to each device register
- use data movement instructions (LD/ST) for control and data transfer



Transfer Timing

- **I/O events generally happen much slower than CPU cycles.**
  - **If** : CPU 300MHz, 10clocks/character,6characters/word
  - **Then:** typing speed $(300x10^6)/(10x6)=5x10^6$words/s
- **Synchronous**
  - **data supplied at a fixed, predictable rate**
  - **CPU reads/writes every X cycles**
- **Asynchronous**
  - **data rate less predictable**
  - **CPU must *synchronize* with device, so that it doesn't miss data or write too quickly**

Transfer Control

- **Who determines when the next data transfer occurs?**
  - CPU vs. I/O device
- **Polling is explicitly looking/examining**
  - CPU keeps checking status register until *new data* arrives OR *device ready* for next data
  - "Are you there yet? Are you there yet? Are you there yet?"
- **Interrupts is a nudge, knock on the door, loud noise, which forces you to pay attention**
  - Device sends a special signal to CPU when *new data* arrives OR *device ready* for next data
  - CPU can be performing other tasks instead of polling device.
  - "Wake me when you get there."

Memory-mapped I/O (Table A.1)

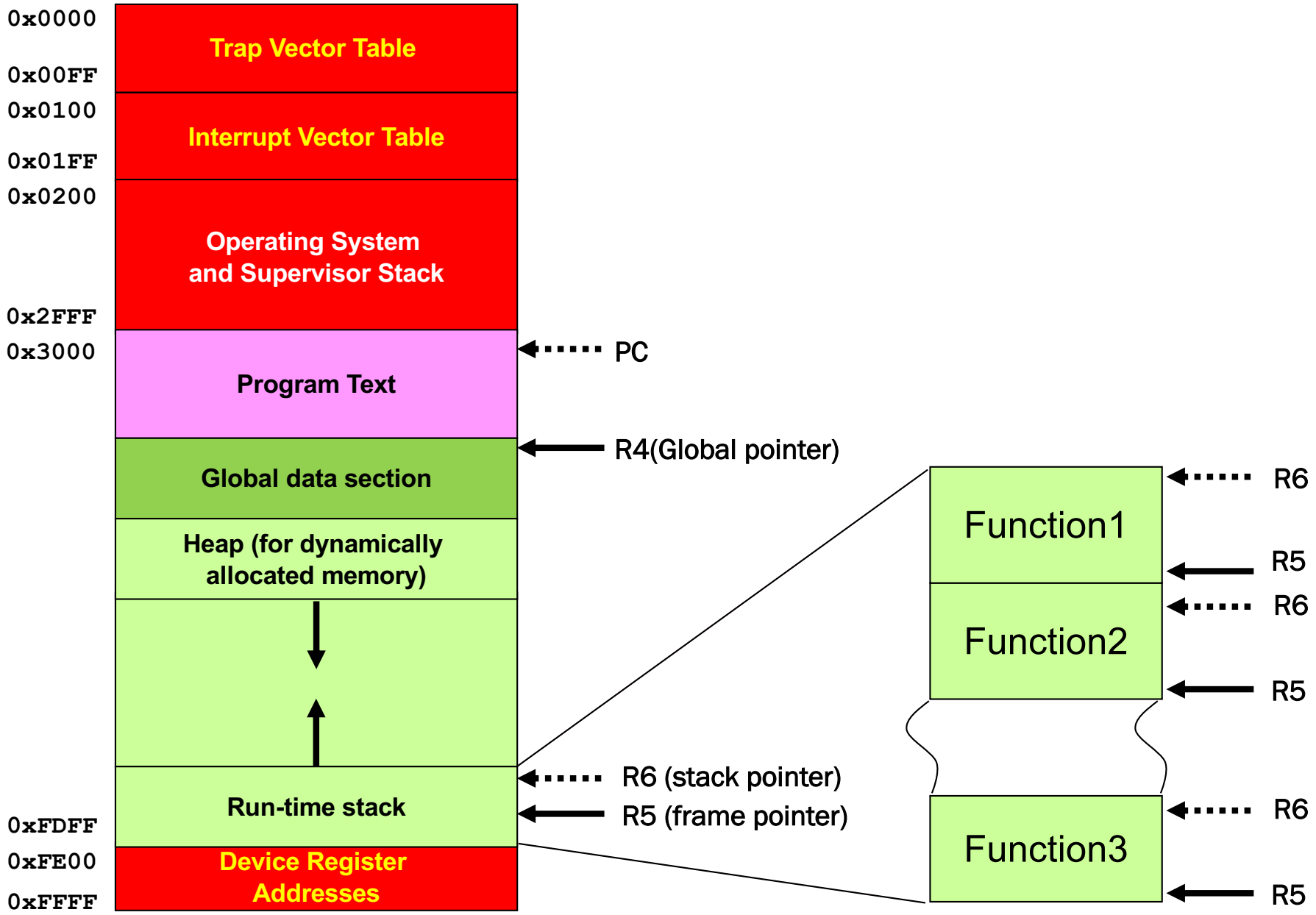
Location	*I/O Register*	*Function*
xFE00	Keyboard Status Reg (KBSR)	Bit [15] is one when keyboard has received a new character.
xFE02	Keyboard Data Reg (KBDR)	Bits [7:0] contain the last character typed on keyboard.
xFE04	Display Status Register (DSR)	Bit [15] is one when device ready to display another char on screen.
xFE06	Display Data Register (DDR)	Character written to bits [7:0] will be displayed on screen.

Asynchronous devices

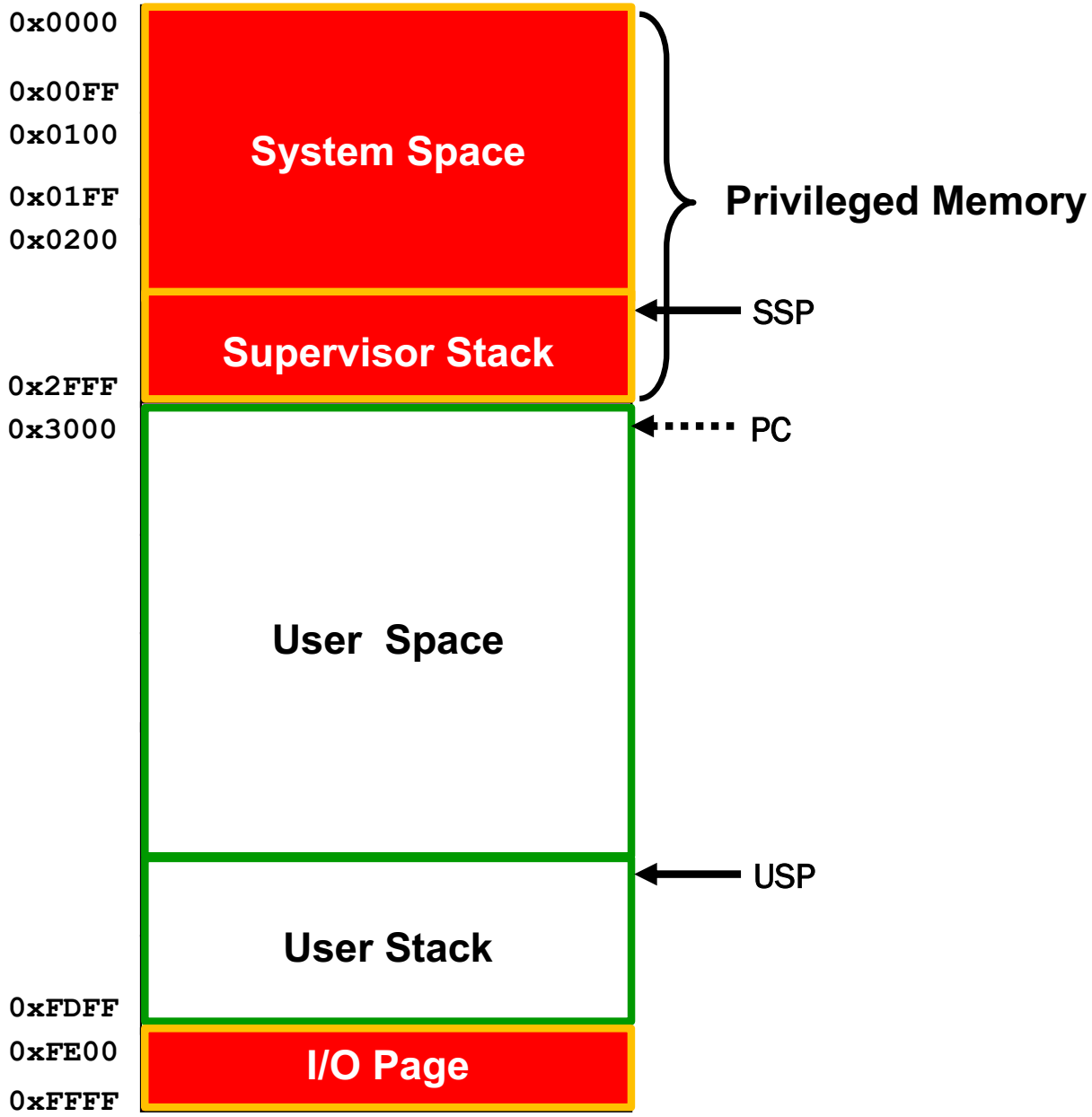
- synchronized through status registers

Polling and Interrupts

Memory Map of the LC-3

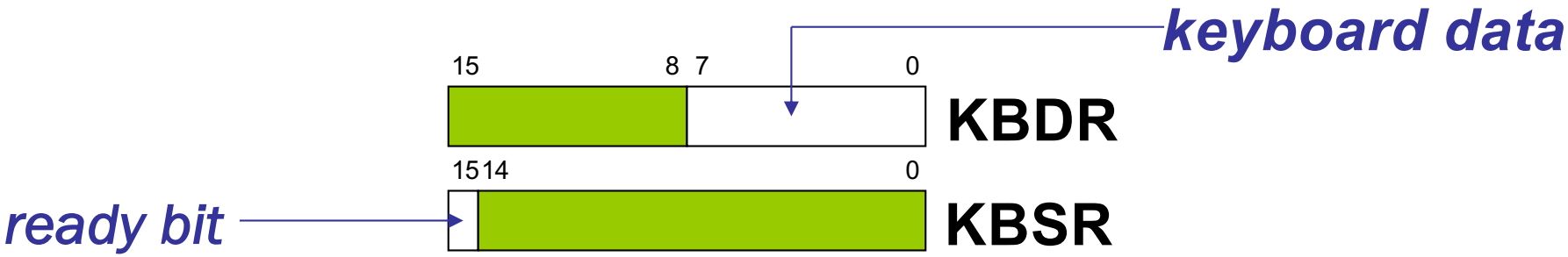


Organization of Memory of the LC-3



Example: Processor Input from Keyboard

- **When a character is typed:**
  - its ASCII code is placed in bits [7:0] of KBDR (bits [15:8] are always zero)
  - the "ready bit" (KBSR[15]) is set to one
  - keyboard is disabled -- any typed characters will be ignored

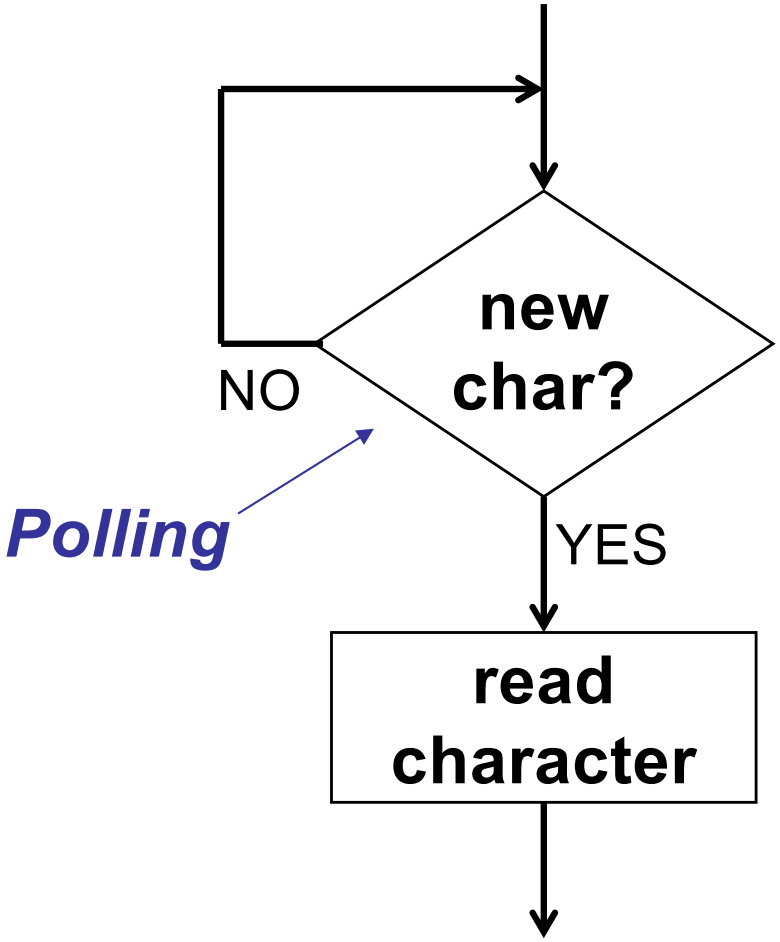


- **When KBDR is read:**
  - KBSR[15] is set to zero, meaning no keyboard key is pending
  - keyboard is enabled

Memory-mapped Operations

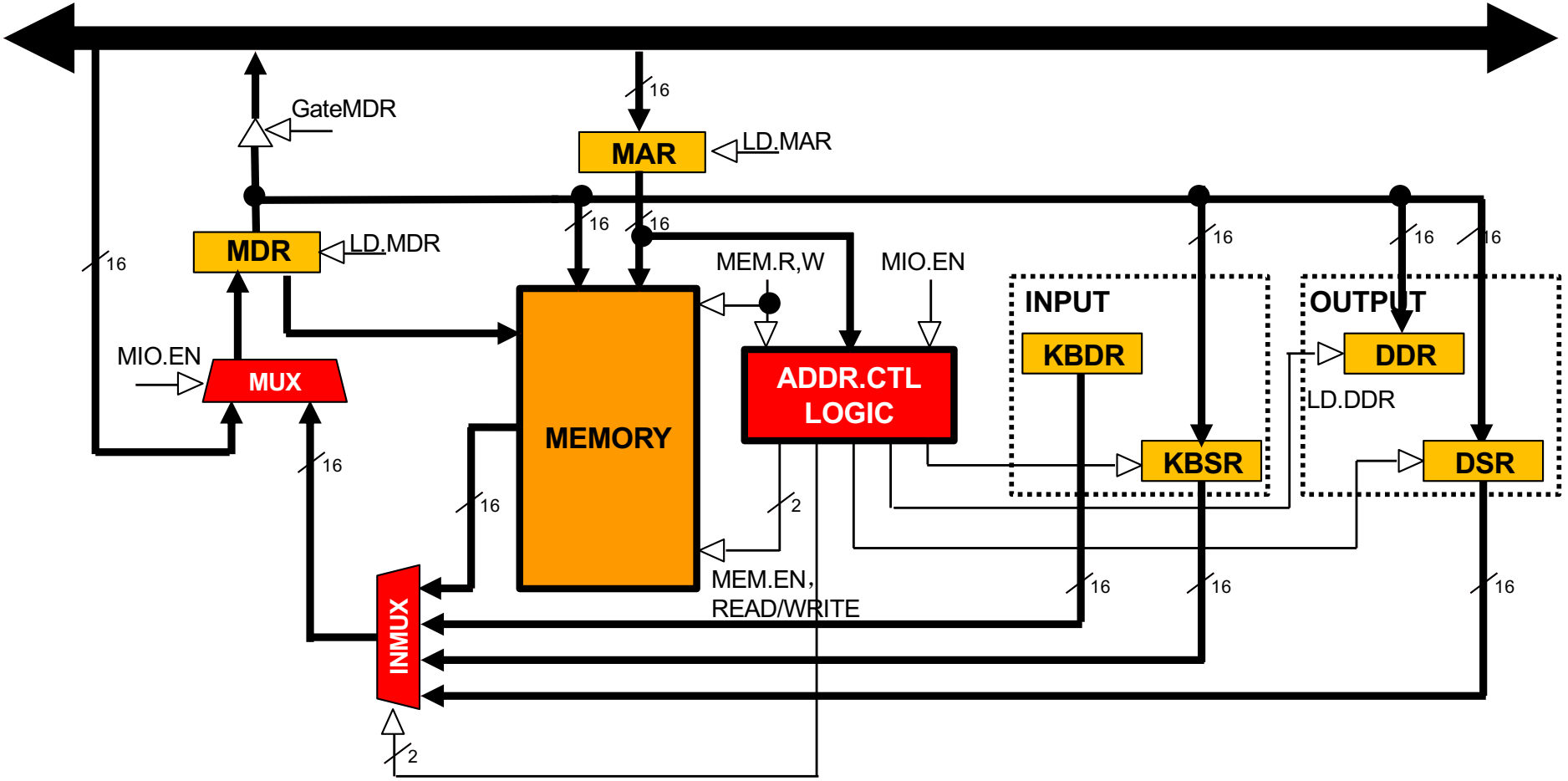
- How do we read ready bit?
- How do we test whether the bit is one?
- How do we read keyboard data?

Basic Input Routine

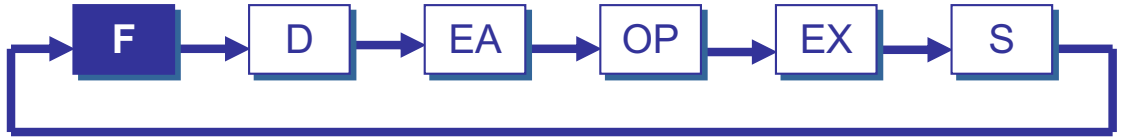



```
POLL    LDI   R0, KBSR
        BRzp  POLL
        LDI   R0, KBDR

        ...

KBSR    .FILL xFE00
KBDR    .FILL xFE02
```

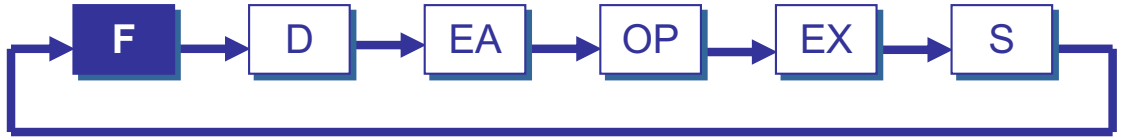
Memory-mapped I/O

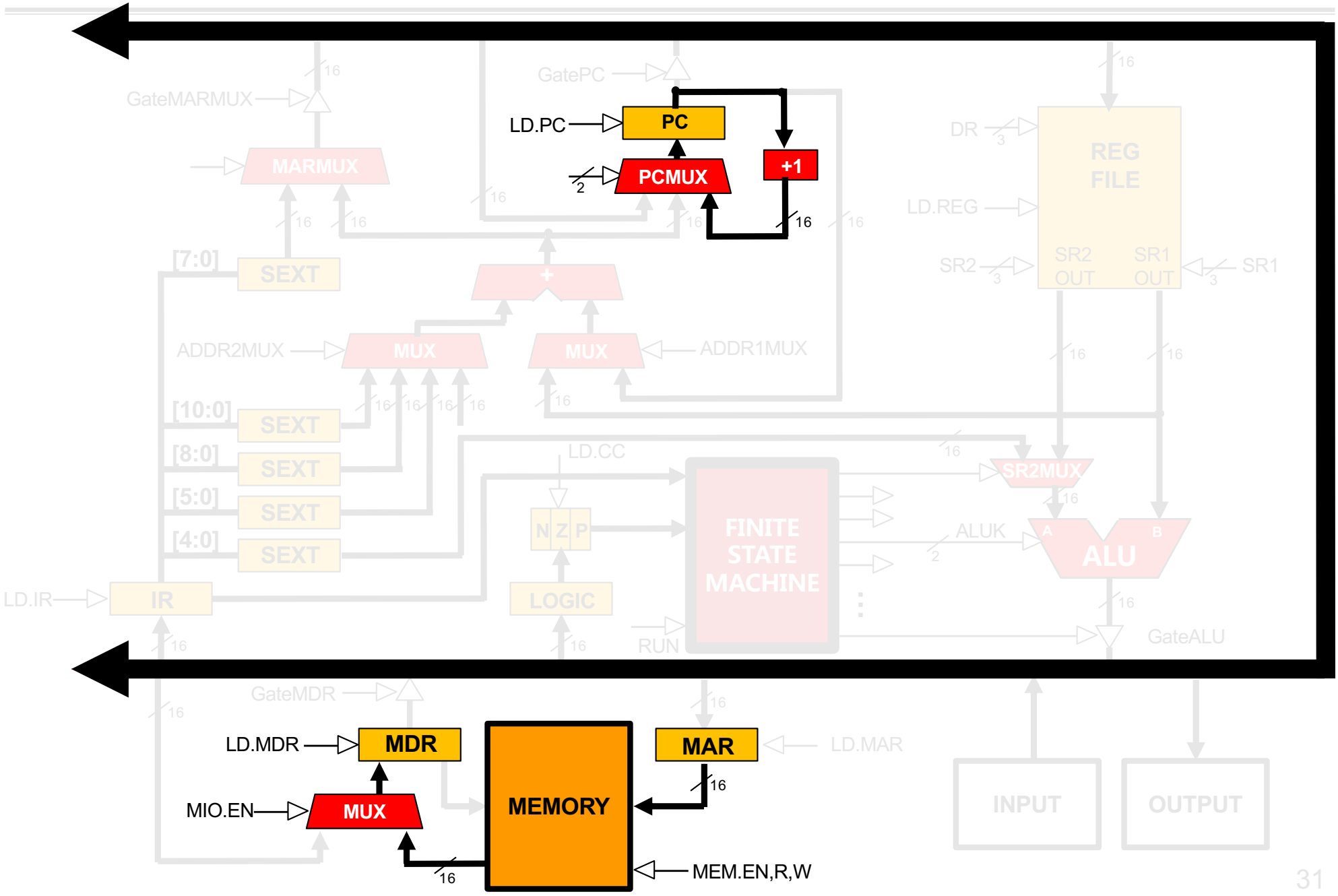


LDI (Indirect)

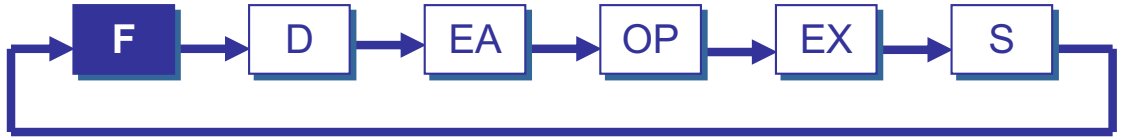


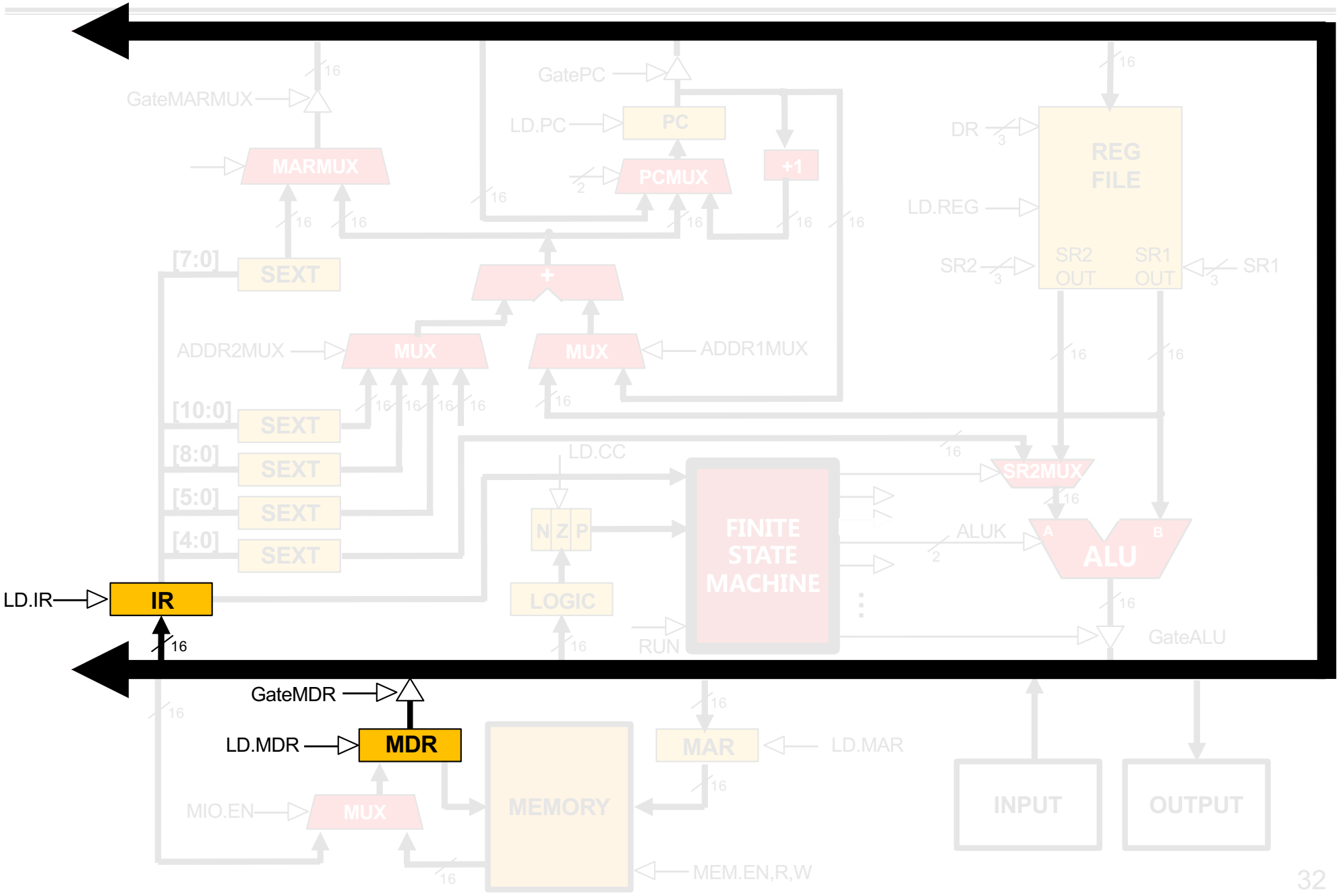
LDI (Indirect)



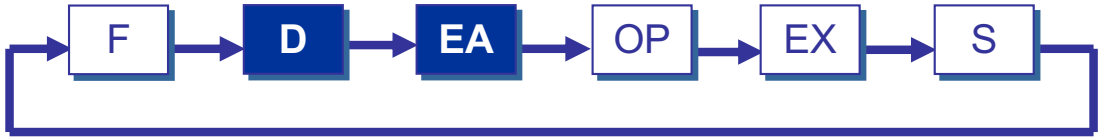


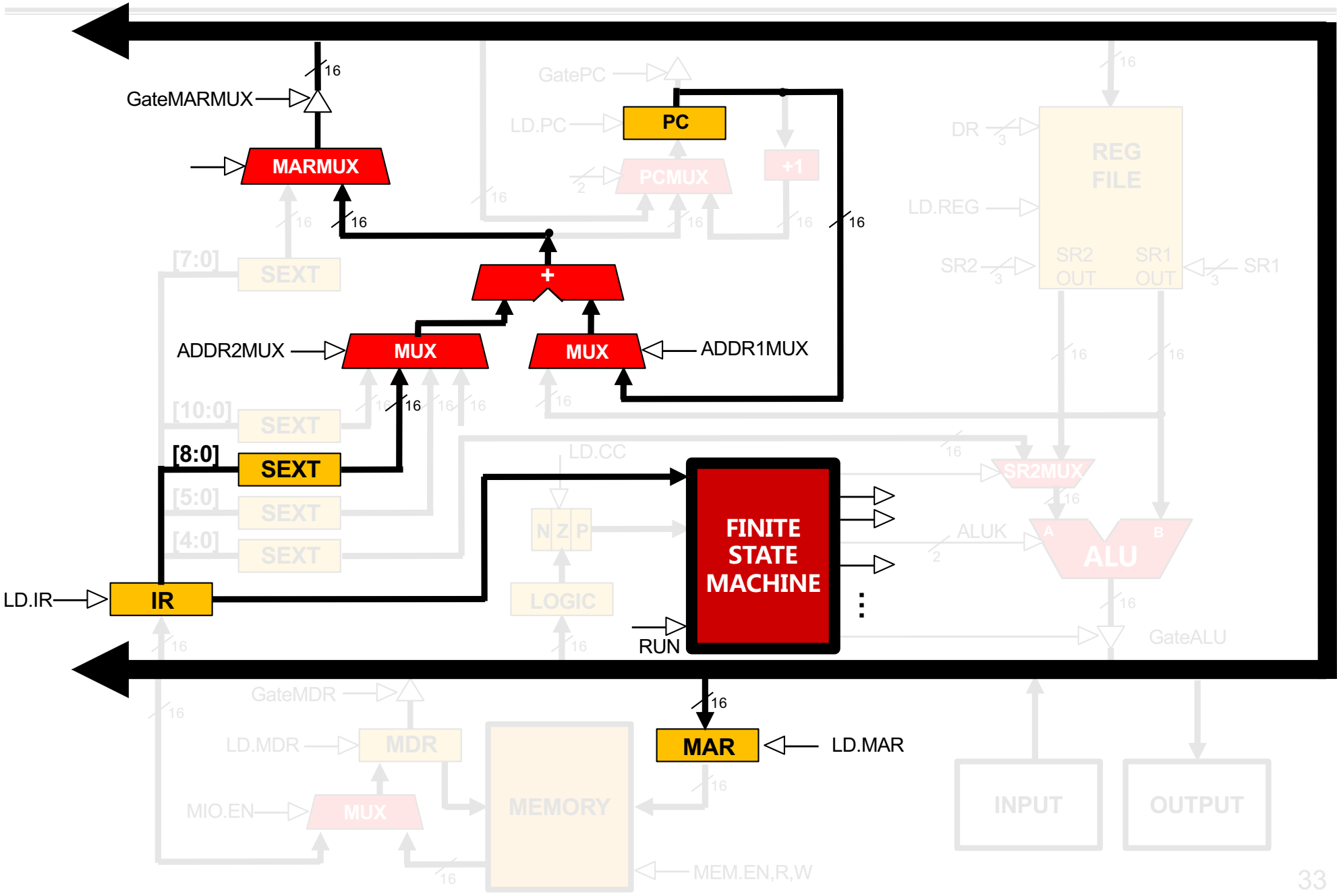
LDI (Indirect)



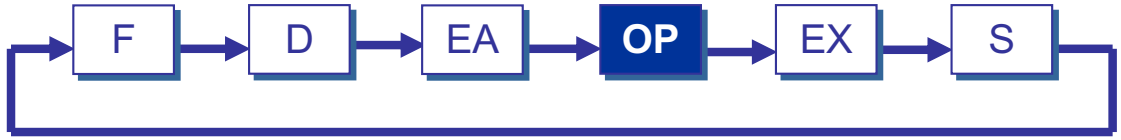


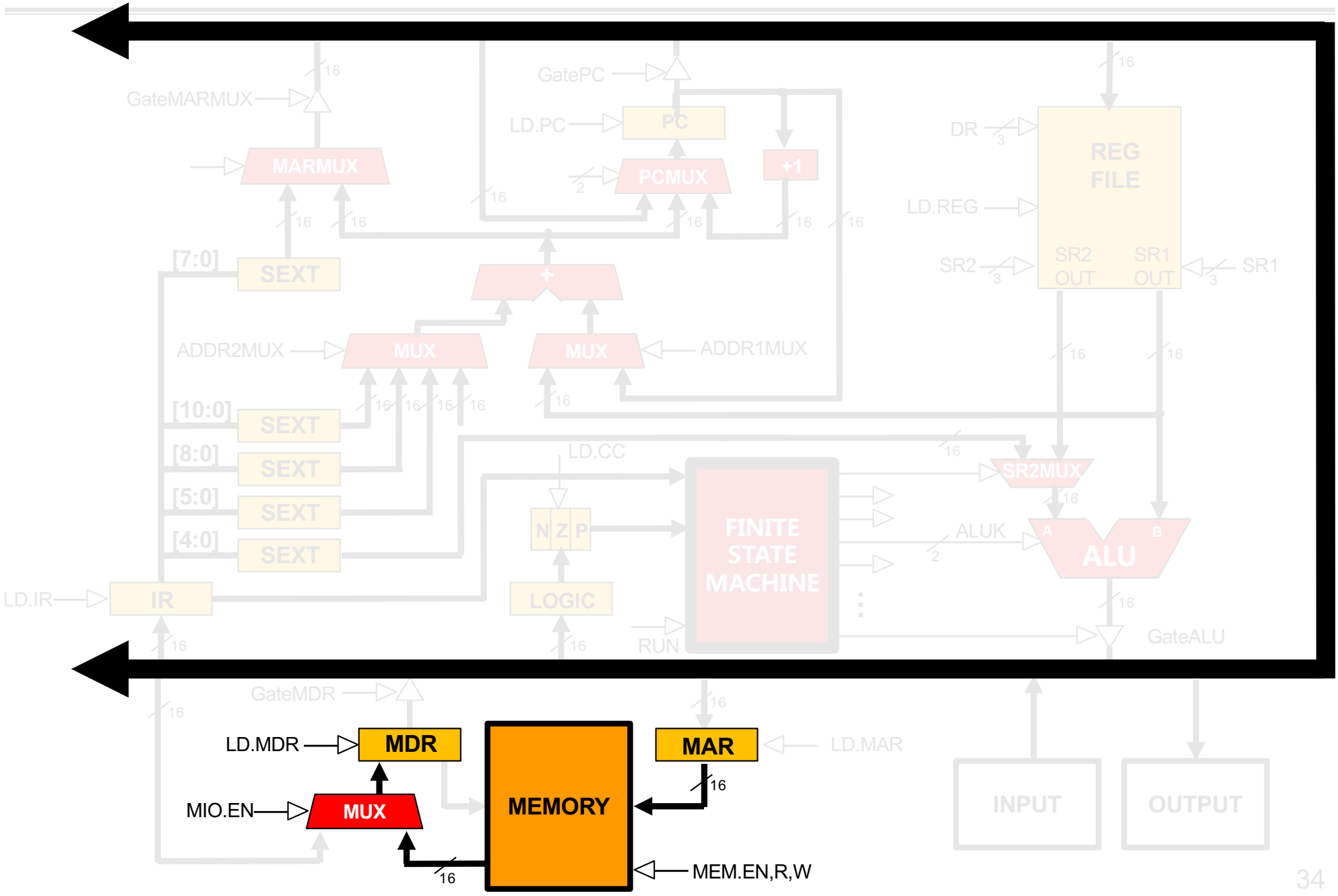
LDI (Indirect)



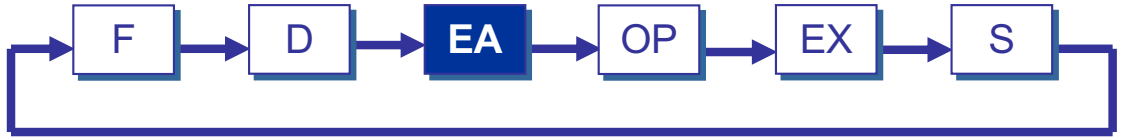


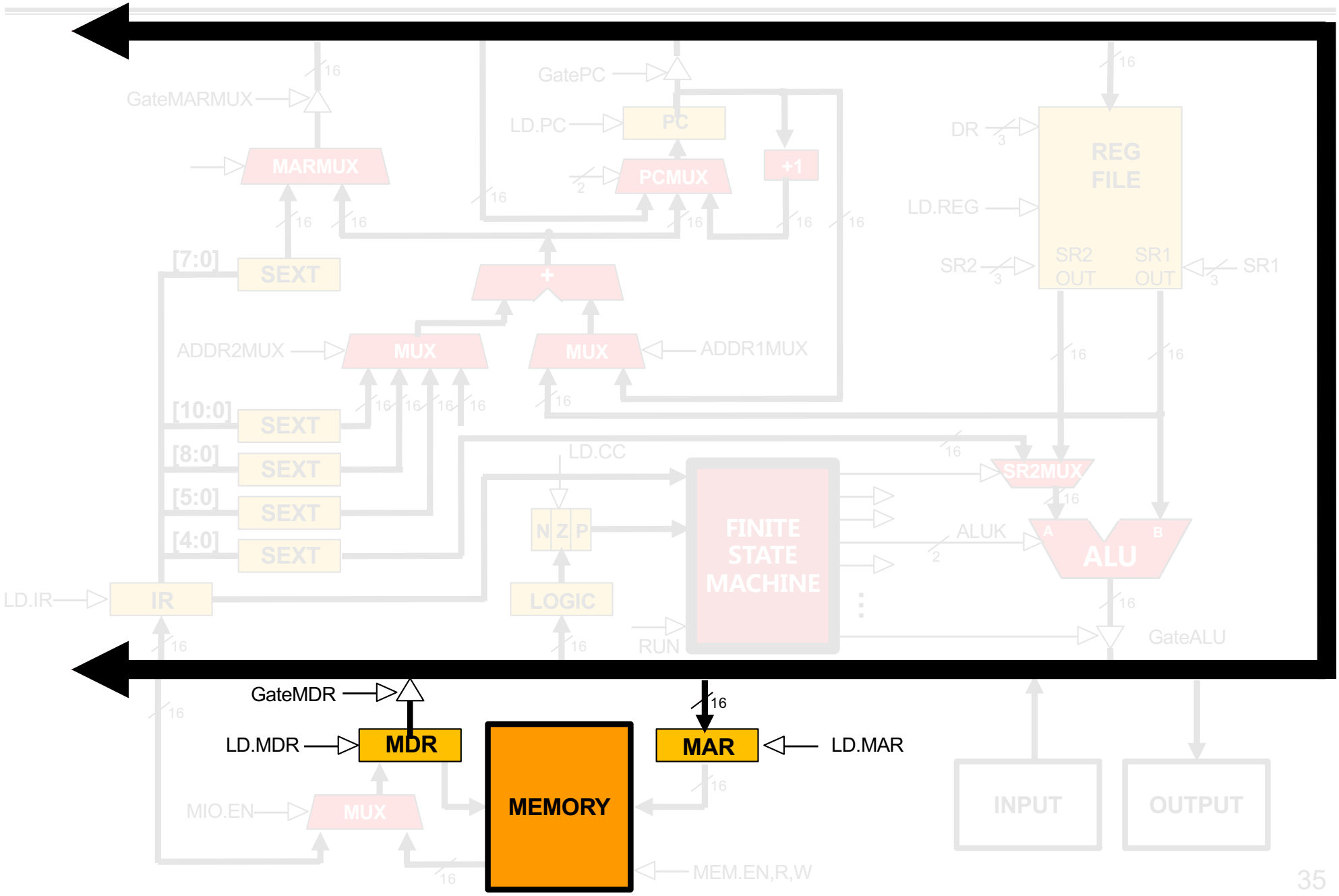
LDI (Indirect)



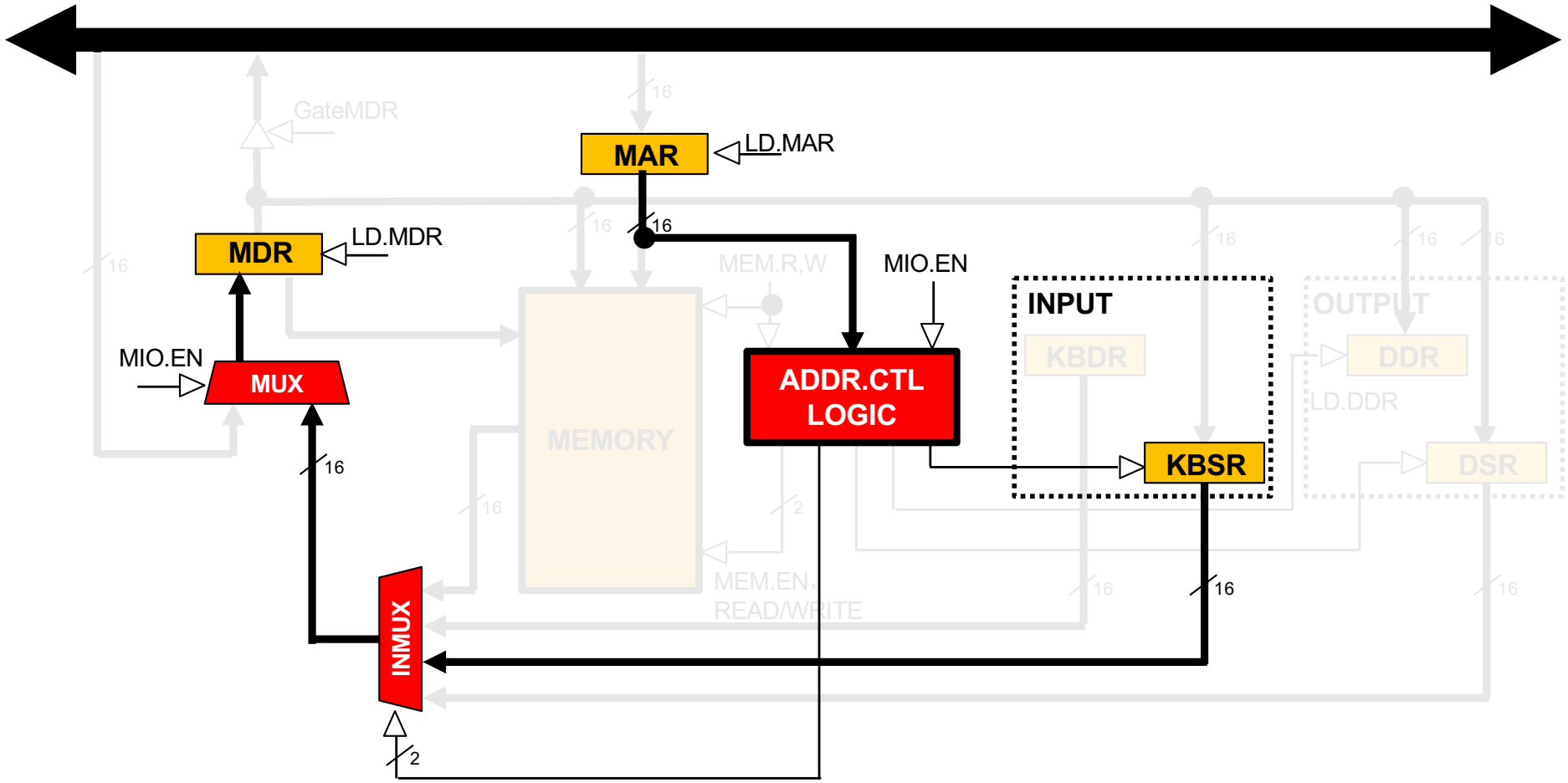


LDI (Indirect)

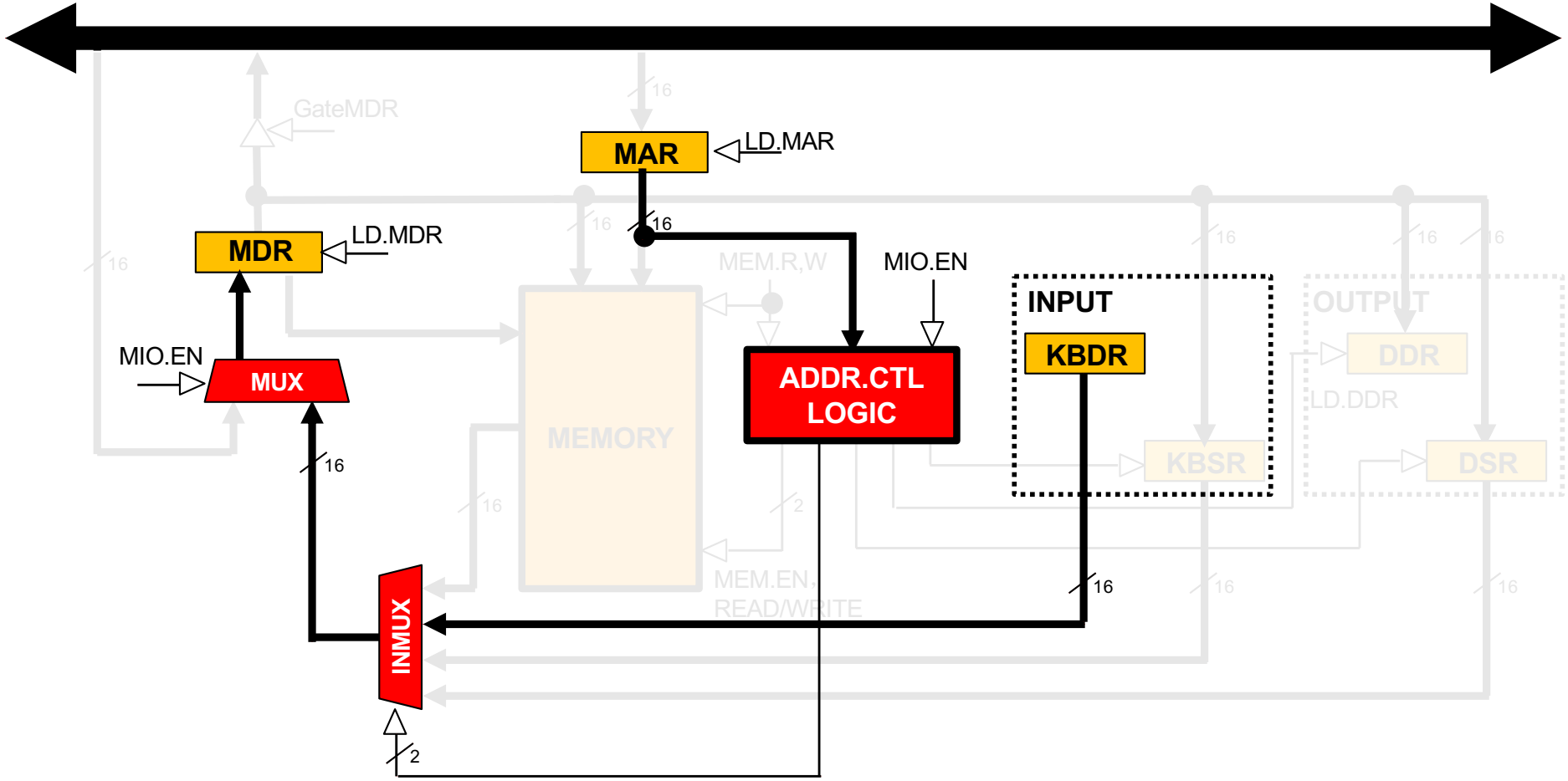




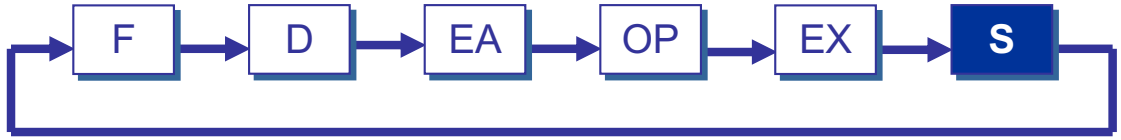
Memory-mapped I/O: `LDI R0, KBSR`

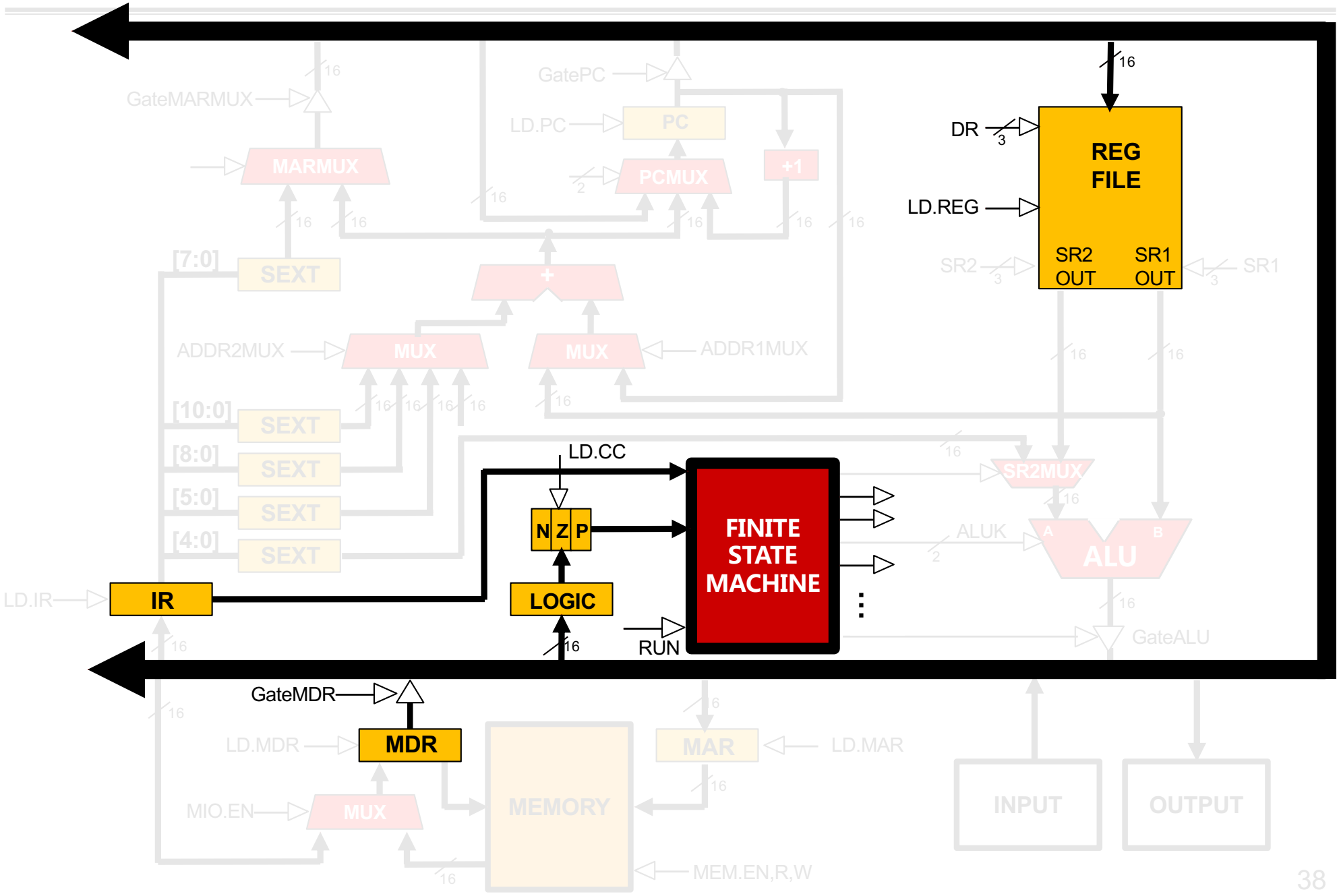


Memory-mapped I/O: `LDI R0, KBDR`



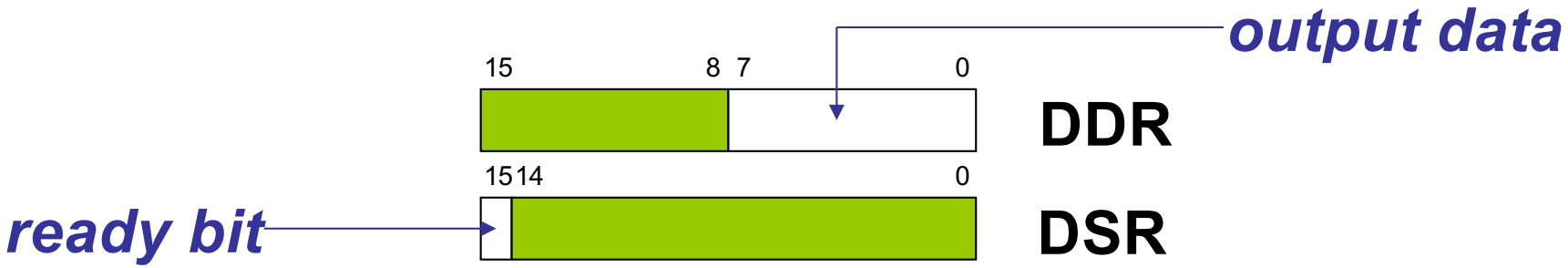
LDI (Indirect)





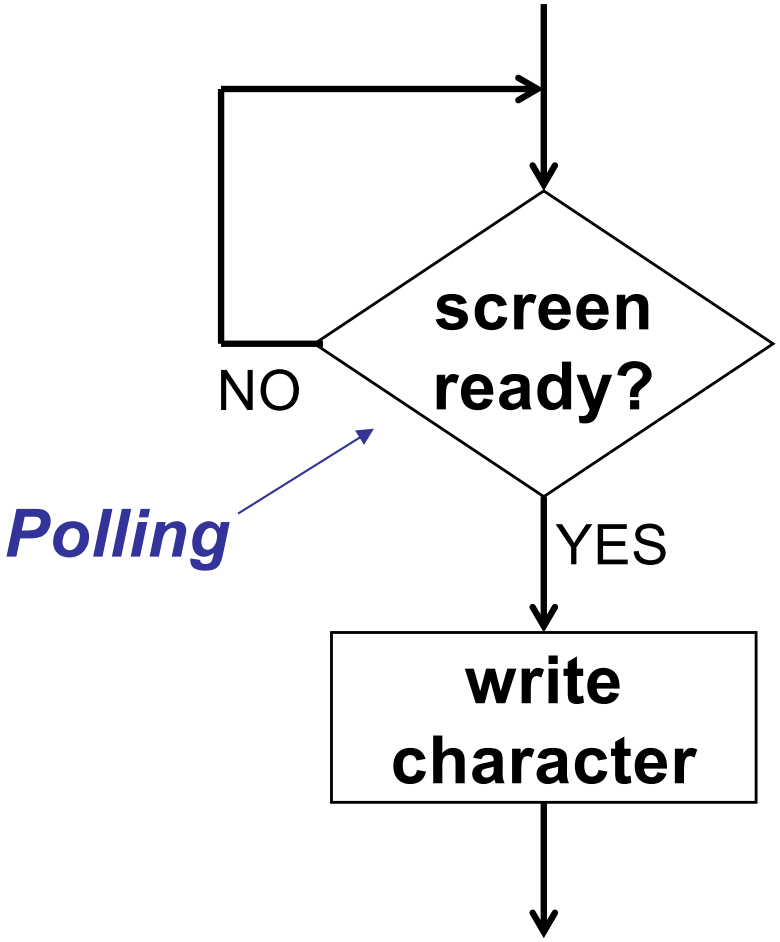
Example: Processor Output to Screen

- When Display device is ready to display another character:
  - the "ready bit" (DSR[15]) is set to one, indicating that processor can write a character for display



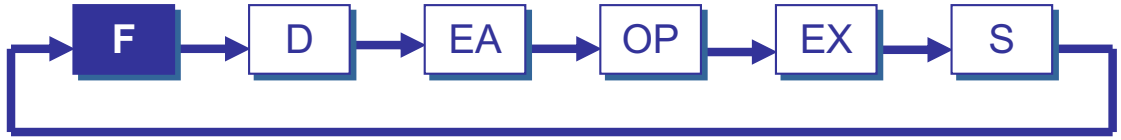
- When data is written to the Display data register:
  - DSR[15] automatically set to 0
  - character in DDR[7:0] is displayed
  - any other character data written to DDR is ignored (while DSR[15] is zero)

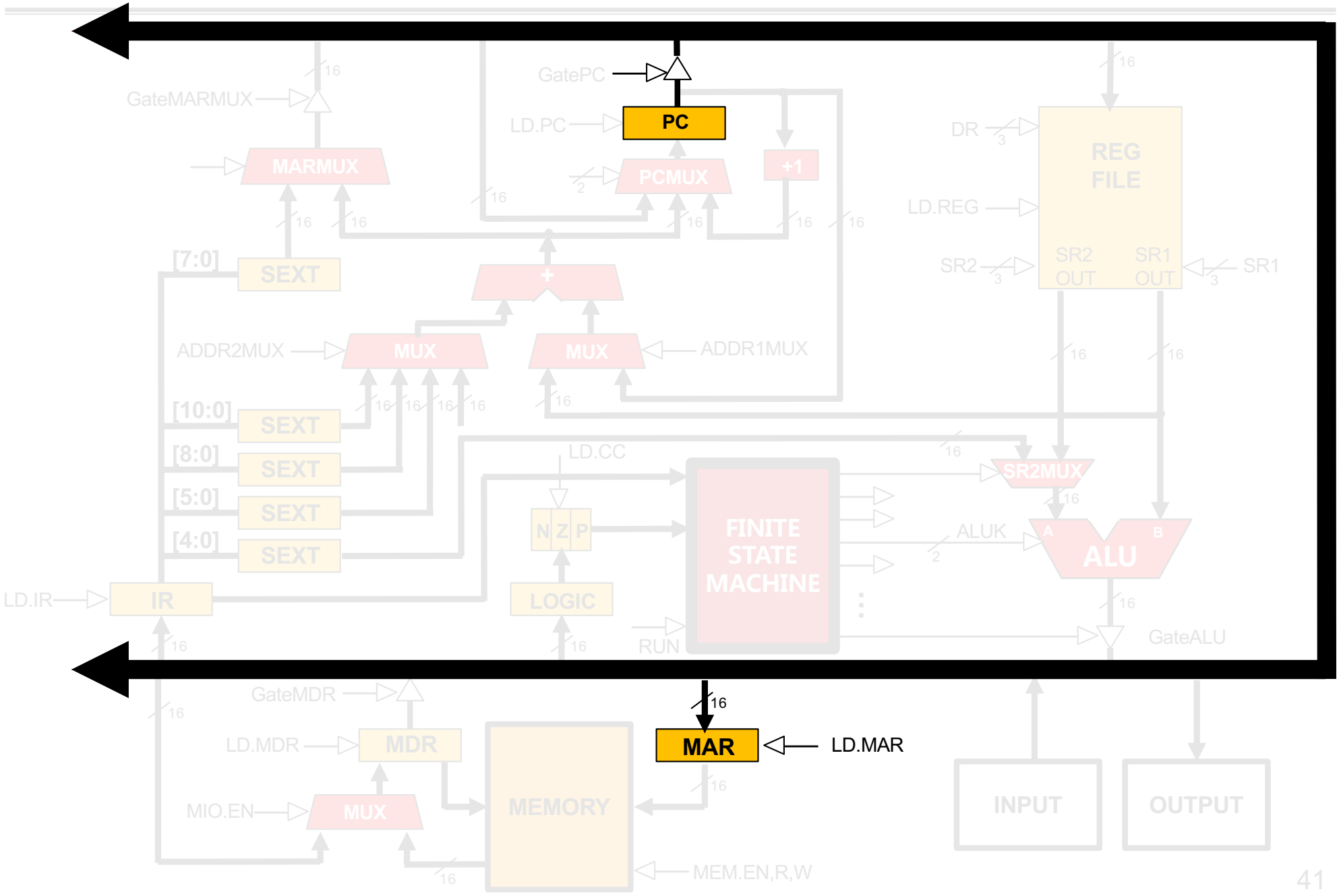
Basic Output Routine



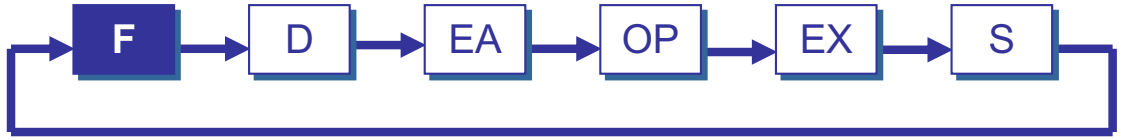

```
POLL    LDI   R1, DSR
        BRzp  POLL
        STI   R0, DDR

        ...

DSR     .FILL xFE04
DDR     .FILL xFE06
```

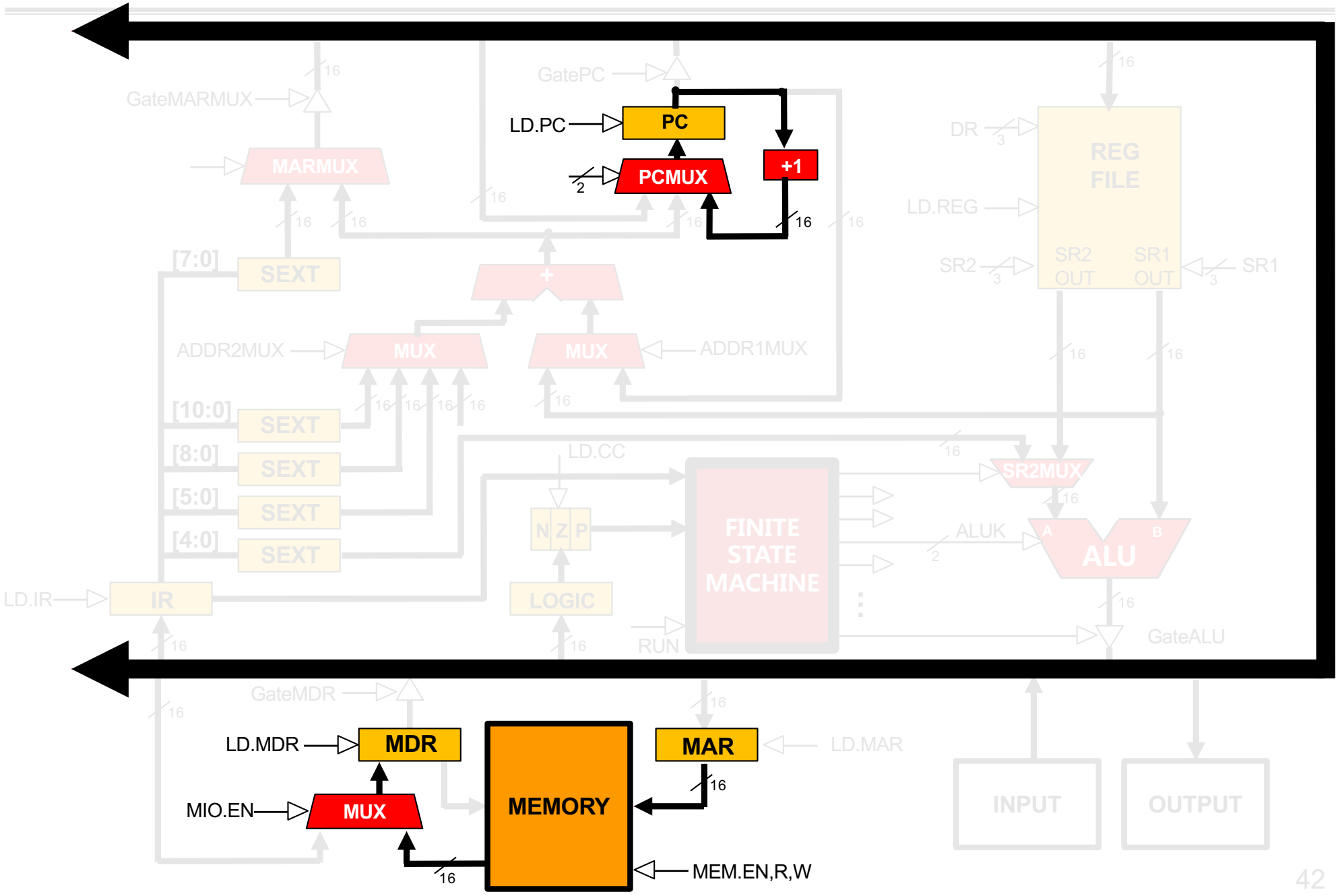
STI (Indirect)



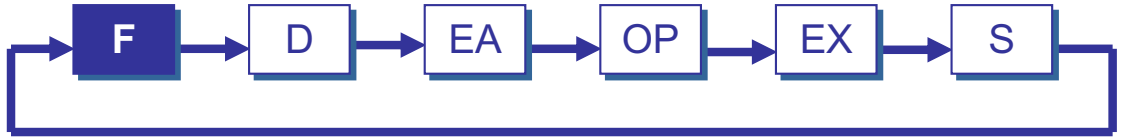


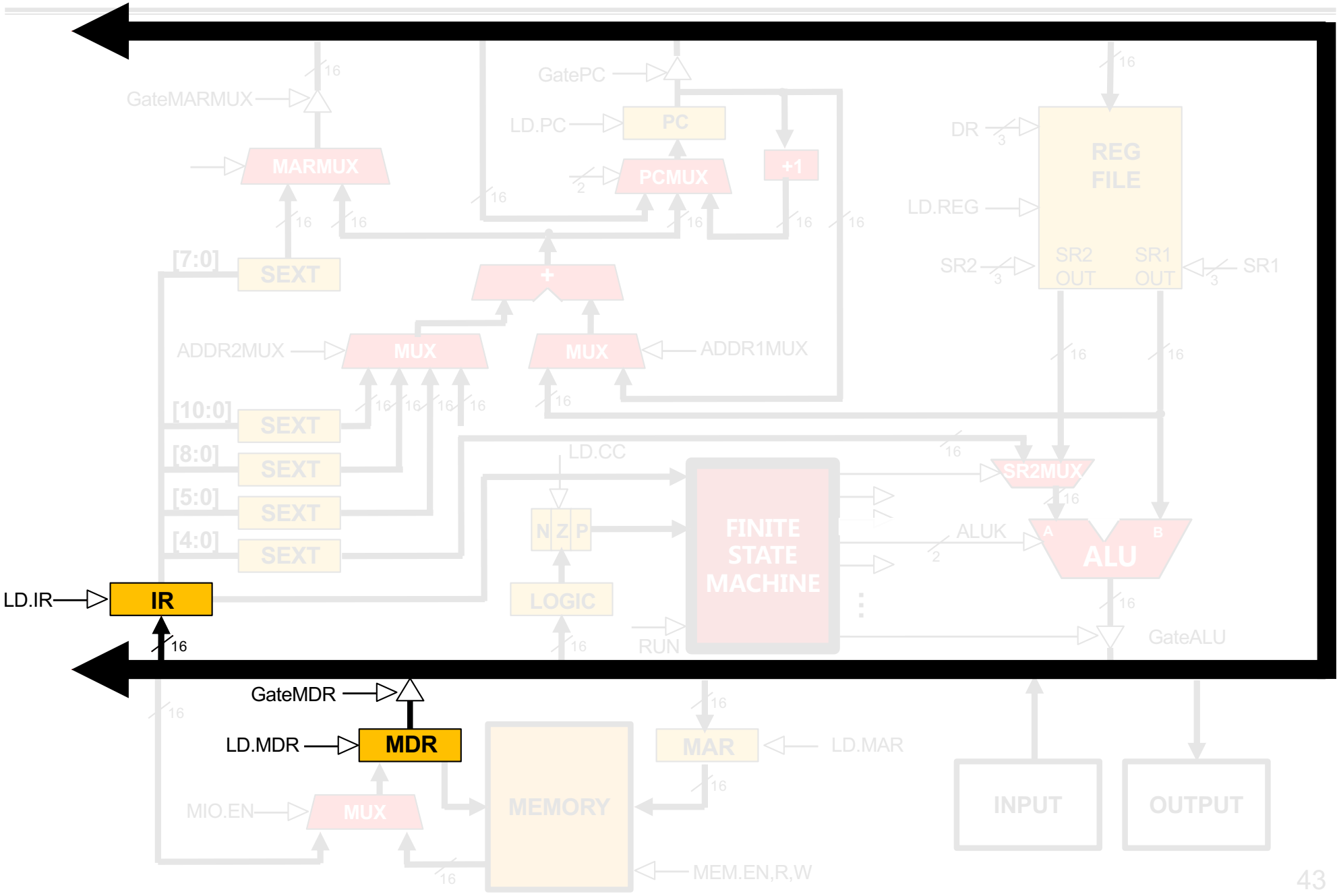
STI (Indirect)



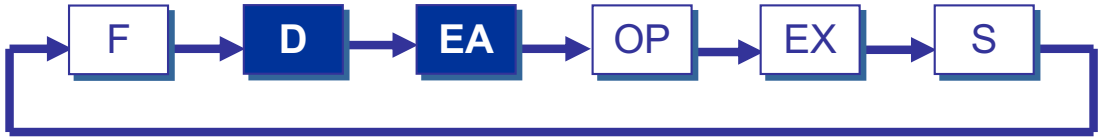


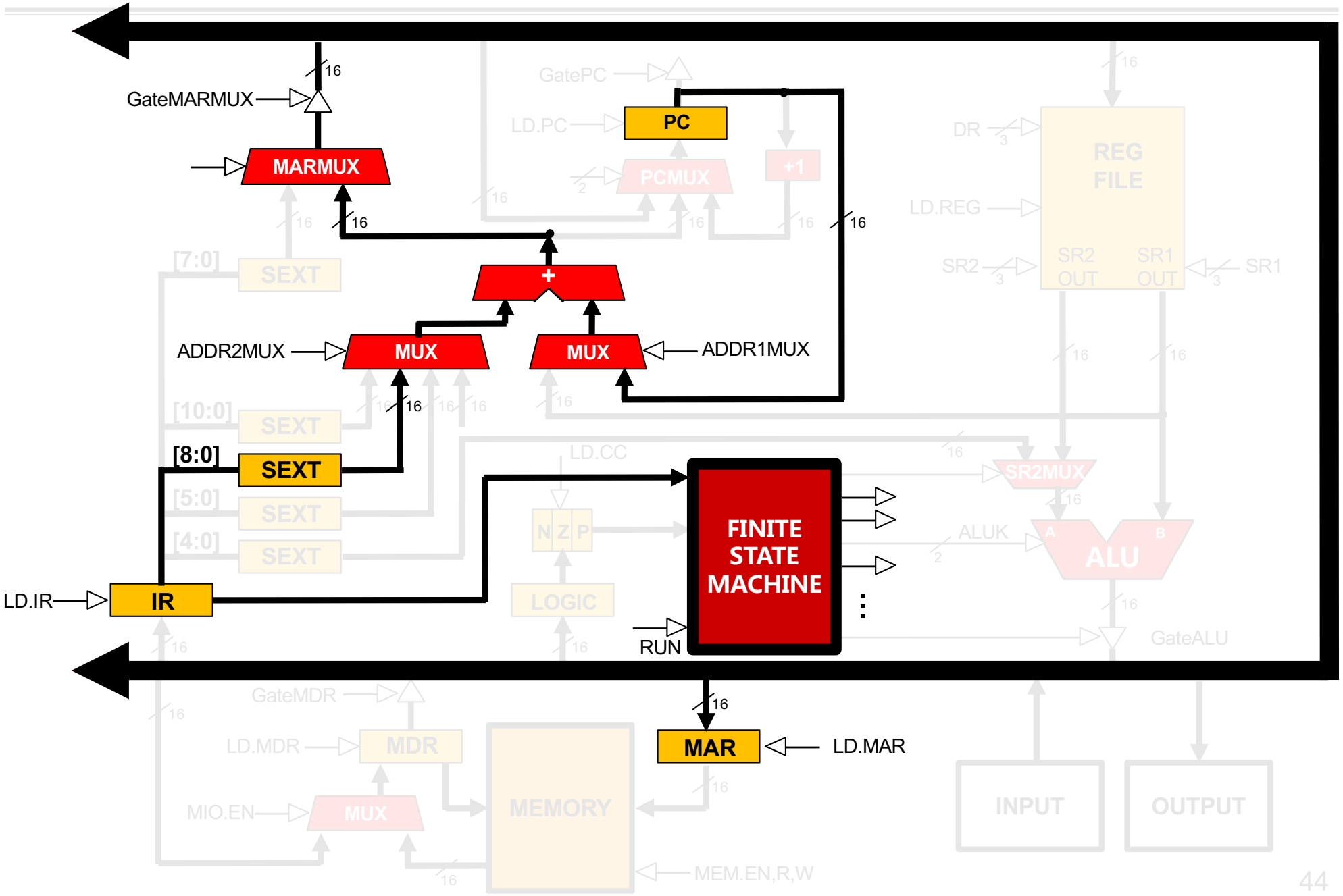
STI (Indirect)



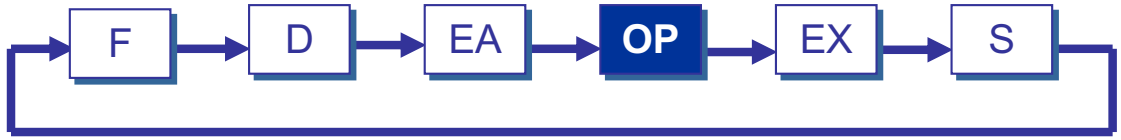


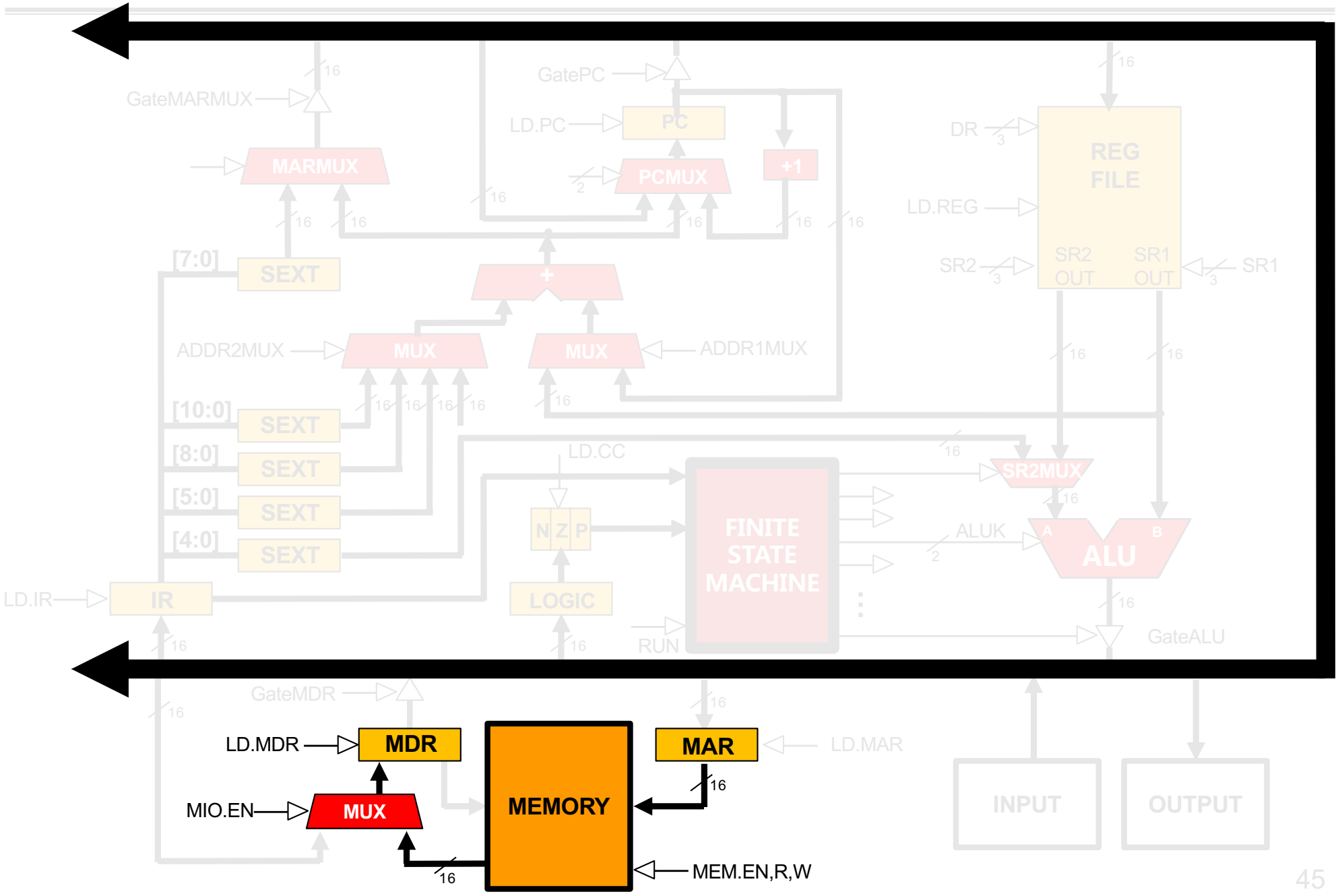
STI (Indirect)



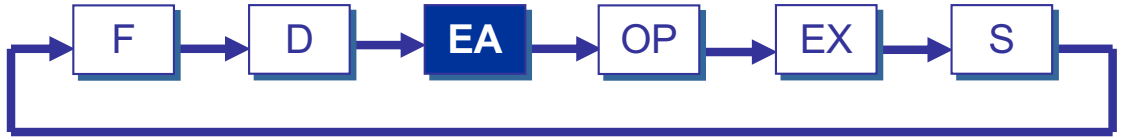


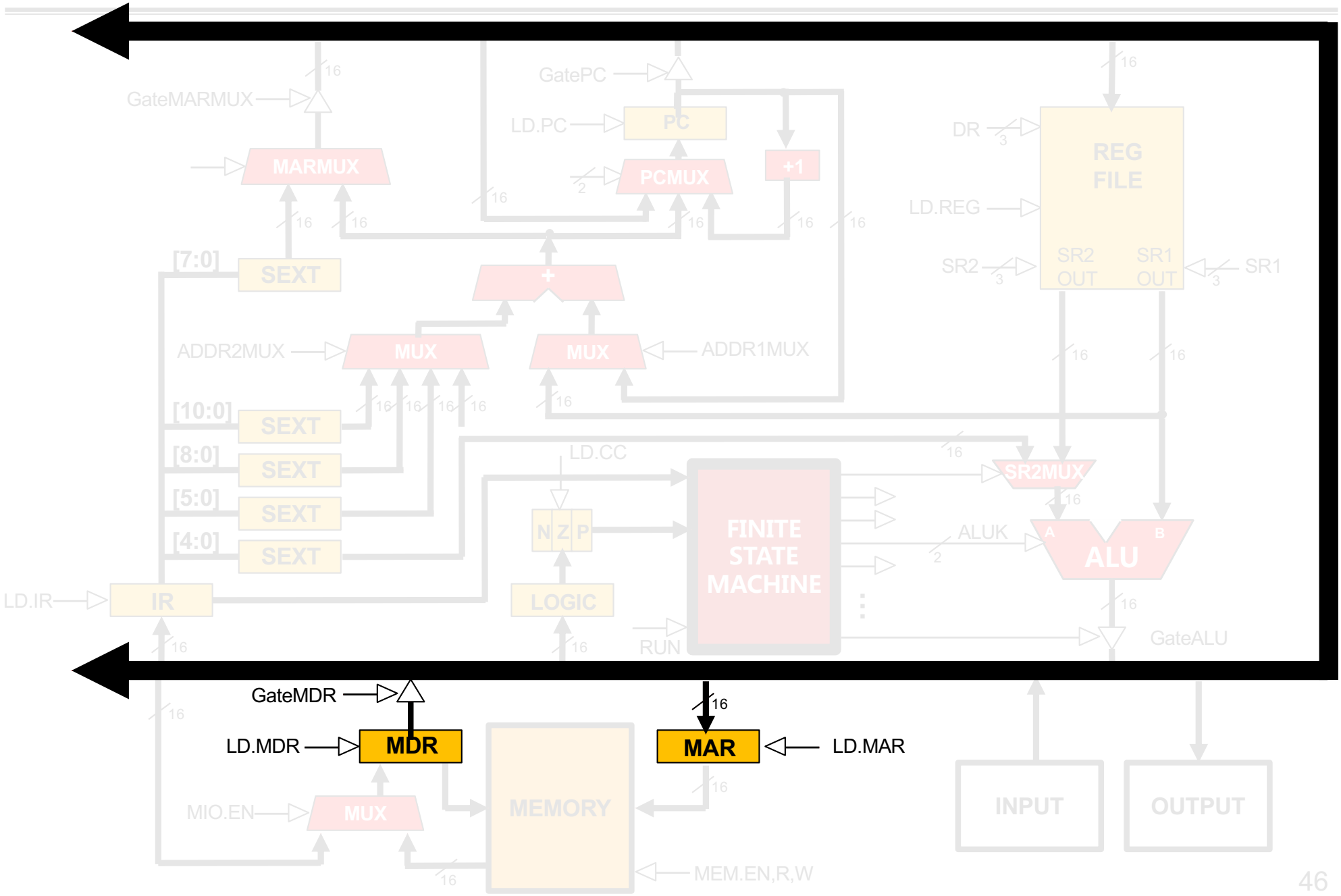
STI (Indirect)



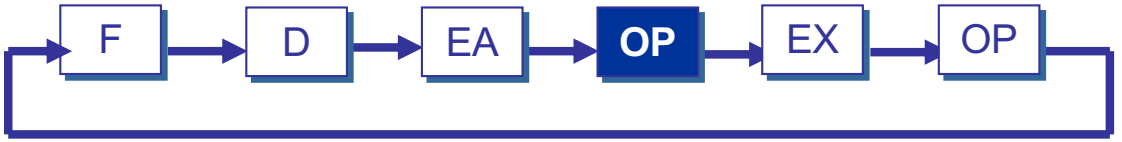


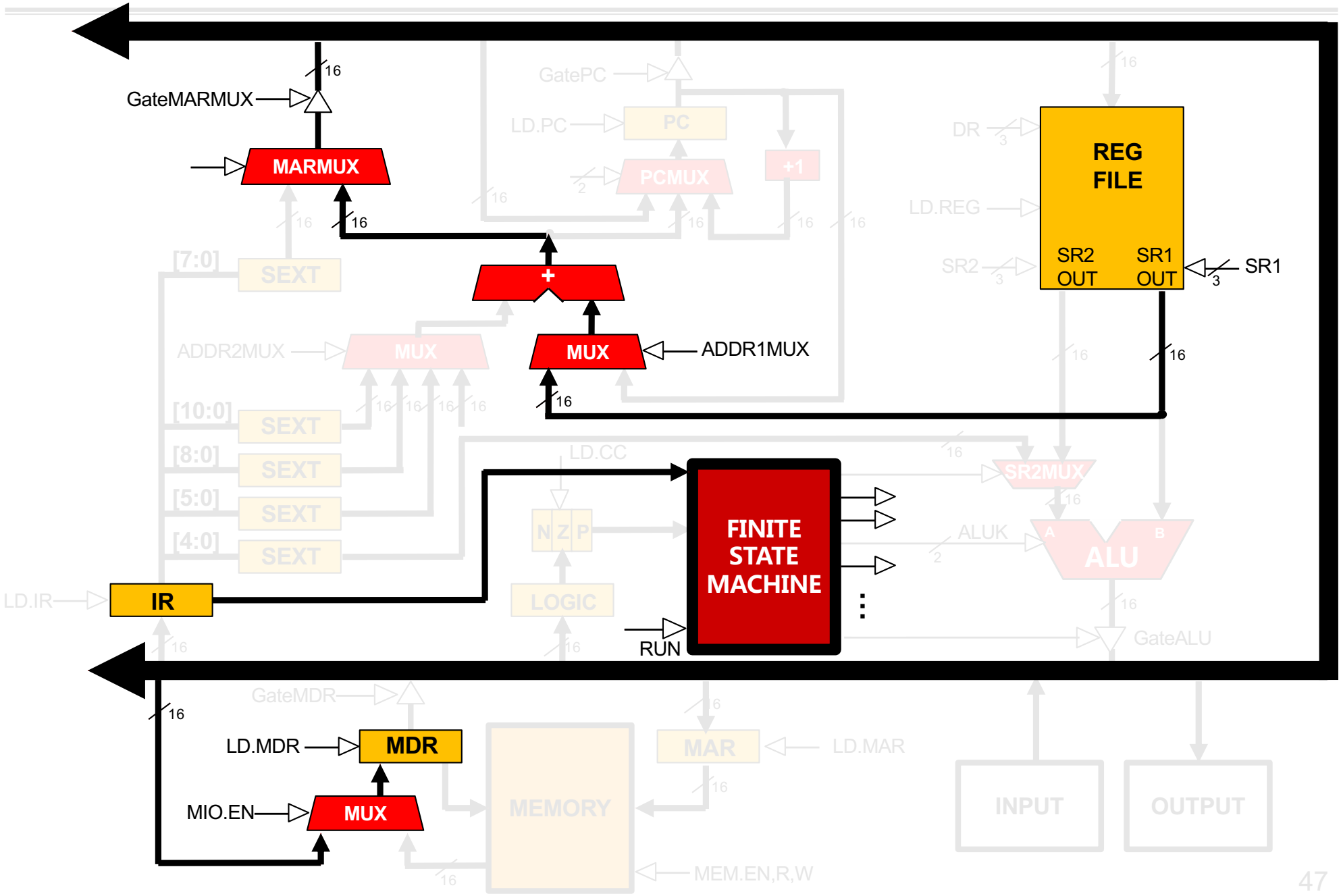
STI (Indirect)



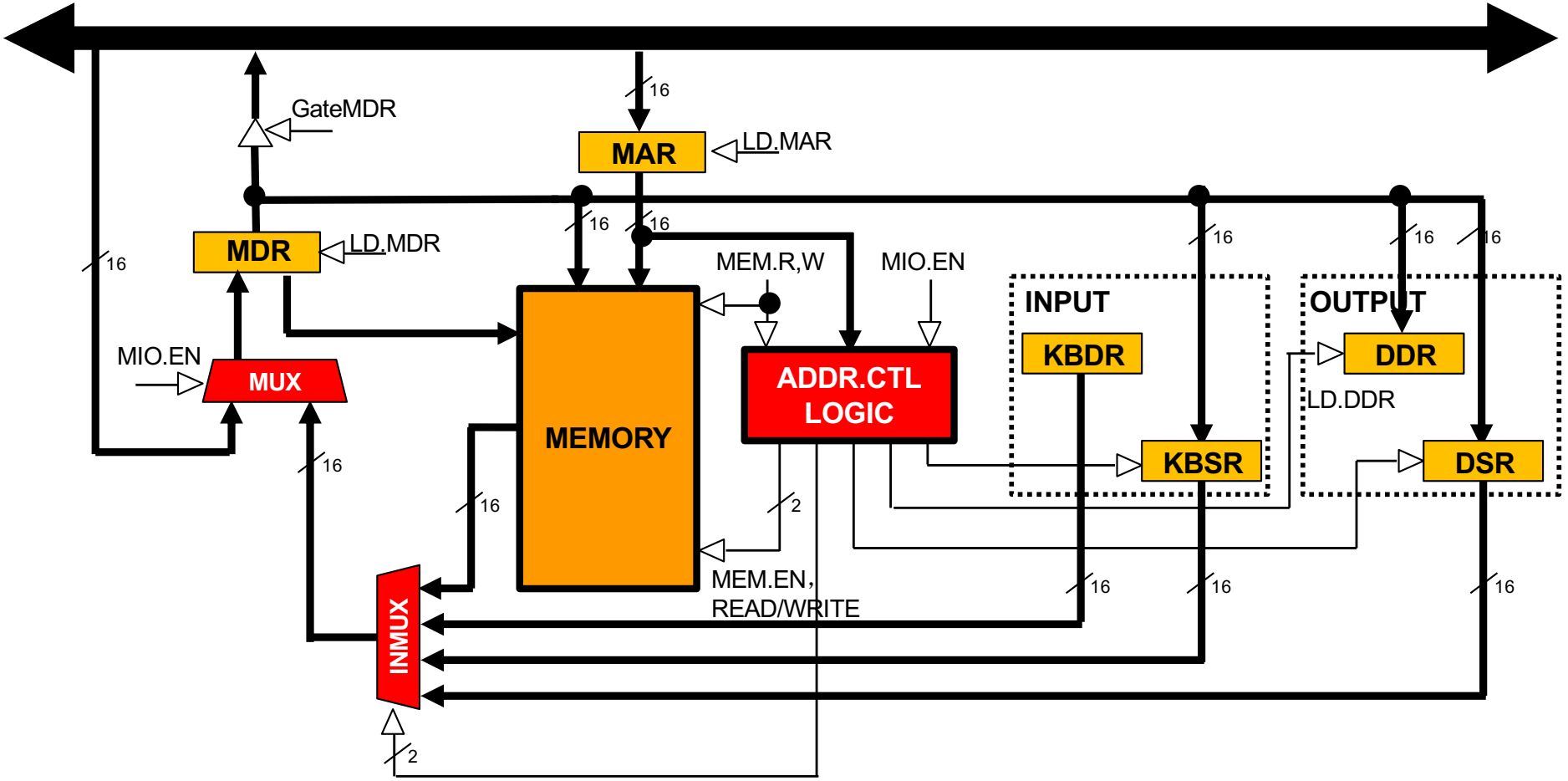


STI (Indirect)





Memory-mapped I/O: STI R0, DDR ?



Keyboard Echo Routine

- **Usually, input character is also printed to screen.**
  - **User gets feedback on character typed and knows its ok to type the next character.**

```
POLL1    LDI   R0, KBSR
         BRzp  POLL1
         LDI   R0, KBDR
POLL2    LDI   R1, DSR
         BRzp  POLL2
         STI   R0, DDR

         ...

KBSR     .FILL xFE00
KBDR     .FILL xFE02
DSR      .FILL xFE04
DDR      .FILL xFE06
```