Computer Organization and Architecture

School of Information Science and Engineering

Southeast University

February 2020

A parallel output controller (POC)

Purpose

The purpose of this project is to design and simulate a parallel output controller (**POC**) which acts an interface between system bus and printer. The Xilinx Vivado is recommended and provided for simulation. Please refer to **William Stallings "Computer Organization and Architecture, Designing for Performance".**

Introduction and requirements

POC is one of the most common **I/O** modules, namely the parallel output controller. It plays the role of an interface between the computer system bus and the peripheral (such as a printer or other output devices).

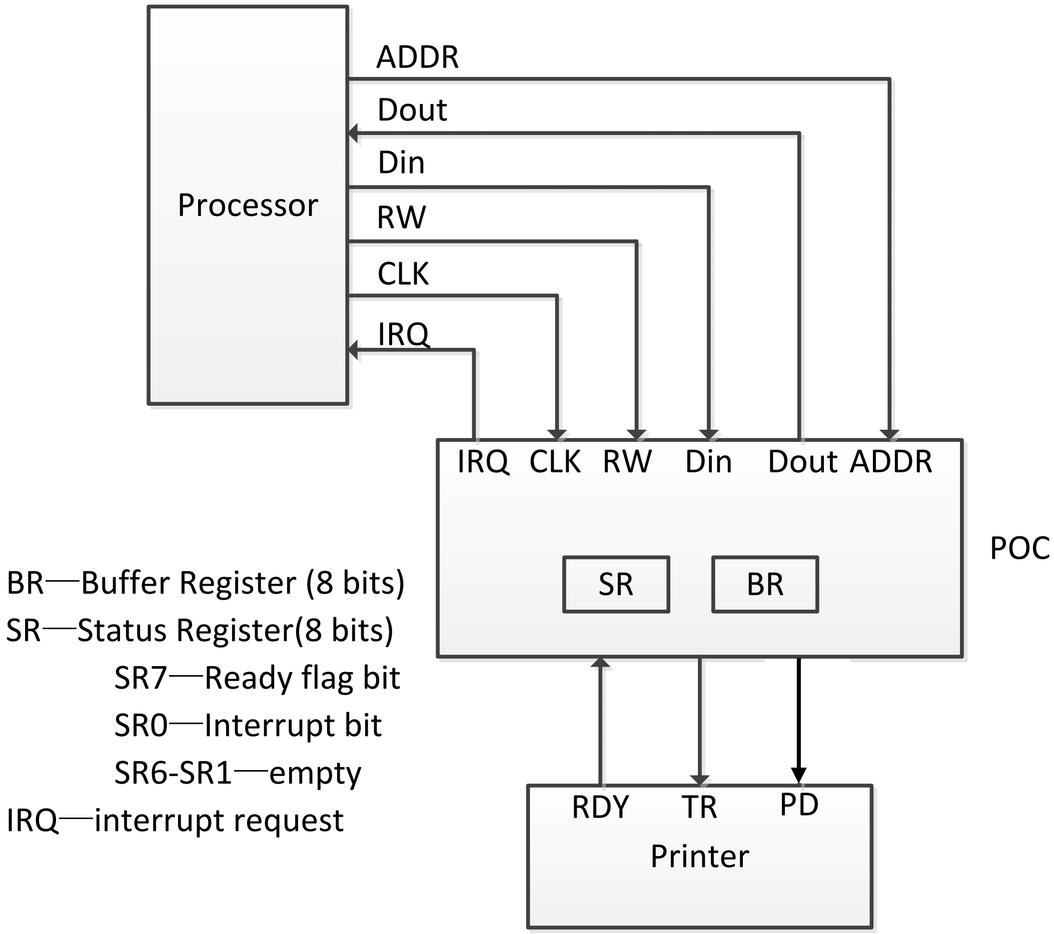

Figure 1 Printer Connection

Figure 1 shows the connecting of a printer to the system bus through the **POC**. The communication between **POC** and the printer is controlled by a "handshake" protocol illustrated in **Figure 2**.

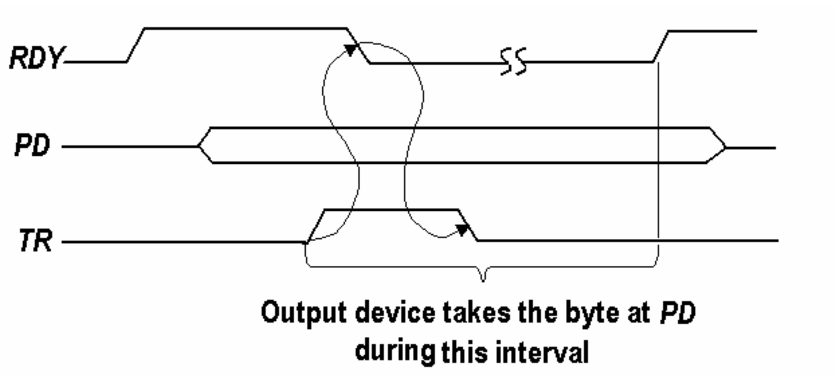

Figure 2 The handshake timing diagram between POC and the printer

The handshaking process is described as follows**:** When the printer is ready to receive a character, it holds **RDY**=**1**.The **POC** then hold a character at **PD** (parallel data) port and generate a pulse at the port **TR** (transfer request). The POC will change TR to 0 when detecting printer has respond TR, i.e., RDY has been changed from 1 to 0. When detecting the effective TR signal, the printer will change **RDY** to **0**, take the character at **PD** and hold the **RDY** at **0** until the character has been printed (e.g. 5 or 10ms), then set **RDY**=**1** again when it is ready to receive the next character. (Suppose the printer has only a one character "buffer" register, so that each character must be printed before the next character is sent).

The buffer register **BR** is used to temporarily hold a character sent from the processor, which character will be transferred to the printer later. The status register **SR** is used for two control functions**: SR7** serves as a ready flag to indicate POC is ready or not to receive a new character from the processor, and **SR0** is used to enable the interrupt requests sent by **POC.** In interrupt mode, If **SR0=1**, then **POC** will send an interrupt request signal to processor when it is ready to receive a character (i.e., when **SR7=1**). If **SR0=0**, then **POC** will not interrupt. The other bits of **SR** are not used and empty.

The transfer of a character to POC via the system bus proceeds as follows.

In polling mode, SR0 is always 0.

The processor selects SR by accessing the relative address, then reads SR register, if SR7=1, the processor selects BR and writes a character into BR, then processor clears SR7 to 0 to indicate that the new character has been written into BR and not printed yet. When POC detects that SR7 is set to 0, POC then proceeds to start the handshaking operations with the printer. After sending character to printer, POC sets the SR7 to 1, which indicates POC is ready to receive another character from the processor. The transfer cycle can now repeat. During the handshaking operations between POC and printer, the processor continues to fetch and execute instructions. If it happens to read SR, it will find SR7=0 and hence will not attempt to send another character to the POC.

In interrupt mode, SR0 is always 1.

After sending character to printer, POC sets the SR7 to 1, since SR0=1, the interrupt request signal (IRQ) is set to 0, which indicate an effective interrupt signal to the processor. When the processor detects the effective IRQ signal, the processor directly selects BR and writes a character into BR, and then the processor sets the SR7 to 0, which indicates that the new character has been written into BR and not printed yet. When POC detects that SR7 is set to 0, POC then proceeds to start the handshaking operations with the printer. After sending character to printer, POC sets the SR7 to 1, which indicates POC is ready to receive another character from the processor. The transfer cycle can now repeat. During the handshaking operations between POC and printer, the processor does not try to access POC until it receives the interrupt request signal.

Requirements of the experiment report

The experiment report should be written in English and following items should be included in it with the same order**:** ①***Title*** of the experiment***;*** ②***Purpose*****;** ③***Tasks*****;** ④***The overall connection of the simulated printer and POC expressed in the top module form***; ⑤***Design description of the simulation input waveforms*****;** ⑥***Simulation results*** (waveforms record and explanation)**;** ⑦***Conclusions and Discussions*****;**

Microprogrammed CPU Design

Purpose

The purpose of this project is to design and verify a simple CPU (Central Processing Unit). This CPU has basic instruction set, and we will utilize its instruction set to generate a very simple program to verify its performance. For simplicity, we will only consider the relationship among the CPU, registers, memory and instruction set. That is to say we only need consider the following items: *Read/Write Registers, Read/Write Memory and Execute the instructions.*

At least four parts constitute a simple CPU: ***the control unit, the internal registers, the ALU and instruction set***, which are the main aspects of our project design and will be studied.

Instruction Set

Single-address instruction format is used in our simple CPU design. The instruction word contains two sections: *the operation code* (opcode), which defines the function of instructions (addition, subtraction, logic operations, etc.); *the address part*, in most instructions, the address part contains the memory location of the datum to be operated, we called it *direct addressing*. In some instructions, the address part is the operand, which is called *immediate addressing*.

For simplicity, the size of memory is 256×16 in the computer. The instruction word has 16 bits. The opcode part has 8 bits and address part has 8 bits. The instruction word format can be expressed in **Figure**

OPCODE	ADDRESS
[15..8]	[7..0]

Figure 3 the instruction format

The opcode of the relevant instructions are listed in **Table 1**.
In **Table 1**, the notation [x] represents the contents of the location x in the memory. For example, the instruction word 0000001110111001_2 ($03B9_{16}$) means that the CPU adds word at location $B9_{16}$ in memory into the accumulator (ACC); the instruction word 0000010100000111_2 (0507_{16}) means if the sign bit of the ACC (ACC [15]) is 0,

the CPU will use the address part of the instruction as the address of next instruction, if the sign bit is 1, the CPU will increase the program counter (PC) and use its content as the address of the next instruction.

Table 1 List of instructions and relevant opcodes

INSTRUCTION	OPCODE	COMMENTS
STORE X	00000001	ACC $\rightarrow$ [X]
LOAD X	00000010	[X] $\rightarrow$ ACC
ADD X	00000011	ACC + [X] $\rightarrow$ ACC
SUB X	00000100	ACC − [X] $\rightarrow$ ACC
JMPGEZ X	00000101	If ACC $\geq$ 0 then X $\rightarrow$ PC else PC+1 $\rightarrow$ PC
JMP X	00000110	X $\rightarrow$ PC
HALT	00000111	Halt a program
MPY X	00001000	ACC $\times$ [X] $\rightarrow$ MR, ACC
AND X	00001010	ACC and [X] $\rightarrow$ ACC
OR X	00001011	ACC or [X] $\rightarrow$ ACC
NOT X	00001100	NOT [X] $\rightarrow$ ACC
SHIFTR	00001101	SHIFT [X] to Right 1bit, Logic Shift
SHIFTL	00001110	SHIFT [X] to Left 1bit, Logic Shift
……	……	……

All the instructions should be implemented in your design.

You must design several programs to test these instructions. A program is given as an example:

Calculate the sum of all integers from 1 to 100.

1. programming with C language:

```
        sum=0;
        temp=100;
  loop :sum=sum+temp;
        temp=temp-1;
        if temp>=0 goto loop;
  end
```

2. Assume in the memory:
   sum is stored at location A4,
   temp is stored at location A3,
   the contents of location A0 is **0**,

the contents of location A1 is **1,**
the contents of location A2 is **$100_{10}=64_{16}$.**

We can translate the above C language program with the instructions listed in **Table 1** into the instruction program as shown in **Table 2**.

Table 2 Example of a program to sum from 1 to 100

Program with C	Program with instructions	Contents of Memory (RAM) in HEX	
		Address	Contents
sum=0;	LOAD A0	00	02A0
	STORE A4	01	01A4
temp=100;	LOAD A2	02	02A2
	STORE A3	03	01A3
loop :sum=sum+temp;	LOOP:LOAD A4	04 (so LOOP=04)	02A4
	ADD A3	05	03A3
	STORE A4	06	01A4
temp=temp-1;	LOAD A3	07	02A3
	SUB A1	08	04A1
	STORE A3	09	01A3
if temp>=0 goto loop;	JMPGEZ LOOP	0A	0504
end	HALT	0B	HALT
		0C	
		…..	…..
		A0	0000
		A1	0001
		A2	0064
		A3	
		A4	
		A5	
		….	…..

You may program to multiply 6 by 5, -6 by 5 and -6 by -5 using MPY instruction, and then check the results.
Note: All data are represented in 2s complement format in memory.

Internal Registers and Memory

MAR *(Memory Address Register)*
MAR contains the memory location of the word to be read from the memory or written into the memory. Here, READ operation is denoted as the CPU reads from memory, and WRITE operation is denoted as the CPU writes to memory. In our design, MAR has 8 bits to access one of 256 addresses of the memory.

MBR *(Memory Buffer Register)*

MBR contains the value to be stored in memory or the last value read from memory. MBR is connected to the address lines of the system bus. In our design, MBR has 16 bits.

PC *(Program Counter)*
PC keeps track of the instructions to be used in the program. In our design, PC has 8 bits.

IR *(Instruction Register)*
IR contains the opcode part of an instruction. In our design, IR has 8 bits.

BR *(Buffer Register)*
BR is used as an input of ALU, it holds other operand for ALU. In our design, BR has 16 bits.

ACC *(Accumulator)*
ACC holds one operand for ALU, and generally ACC holds the calculation result of ALU. In our design, ACC has 16 bits.

MR *(Multiplier Register)*
MR is used for implementing the MPY instruction, holding the multiplier at the beginning of the instruction. When the instruction is executed, it holds part of the product.

DR *(Division Register)*
DR is possibly used for implementing the DIV instruction, you can define it according to your division algorithm.

RAM
RAM with separate input and output ports, it works as memory which stores the instructions and data, and its size is 256×16. Although it's not an internal register of CPU, we need it to simulate and test the performance of CPU.

All the registers are positive-edge-triggered.
All the reset signals for the registers are synchronized to the clock signal.

ALU

ALU (Arithmetic Logic Unit) is a calculation unit which accomplishes basic arithmetic and logic operations. In our design, some operations must be supported which are listed as follows

Table 3 ALU Operations

Operations	Explanations
ADD	(ACC)$\Leftarrow$(ACC) + (BR)
SUB	(ACC)$\Leftarrow$(ACC) − (BR)
AND	(ACC)$\Leftarrow$(ACC) and (BR)
OR	(ACC)$\Leftarrow$(ACC) or (BR)
NOT	(ACC)$\Leftarrow$Not (ACC)
SRL	(ACC)$\Leftarrow$Shift (ACC) to Left 1 bit
SRR	(ACC)$\Leftarrow$Shift (ACC) to Right 1 bit

Microprogrammed Control Unit

We have learnt the knowledge of Microprogrammed control unit. Here, we only review some terms and basic structures.

In the Microprogrammed control, the microprogram consists of some microinstructions and the microprogram is stored in control memory that generates all the control signals required to execute the instruction set correctly. The microinstruction contains some micro-operations which are executed at the same time.

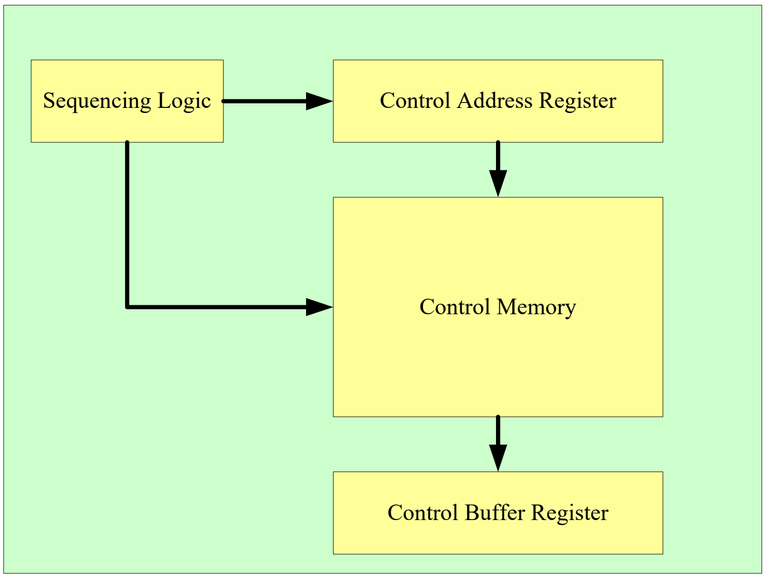

Figure 4 Control Unit Micro-architecture

Figure shows the key elements of such an implementation. The set of microinstructions is stored in the control memory. The control address register contains the address of the next microinstructions to be read. When a microinstruction is read from the control memory, it is transferred to a control buffer register. The register connects to the control lines emanating from the control unit. Thus, reading a microinstruction from the control memory is the same as executing that microinstruction. The third element shown in the figure is a sequencing unit that loads the control address register and issues a read command.

CPU Design

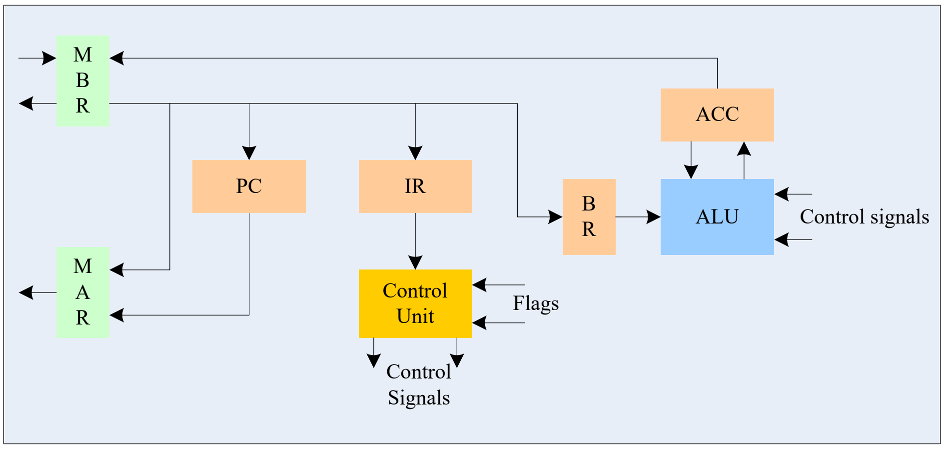

Figure 5 CPU data path and control signals

Figure indicates a simple CPU architecture and its use of a variety of internal data paths and control signals. Our CPU design should be based on this architecture.

You should determine the control signals according to the CPU architecture and your design. An example is given below to show the procedure, this example describes the control unit design for the ***LOAD*** instruction.

First, we need determine the control flowchart of the ***LOAD*** instruction.

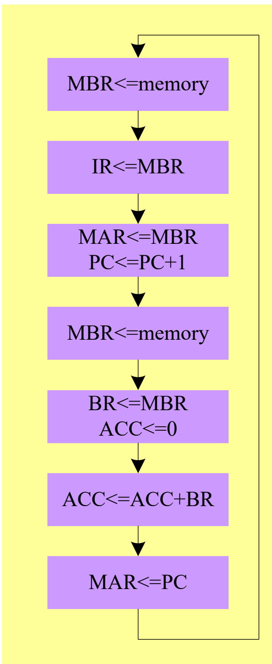

Figure 6 Control Flowchart of the LOAD instruction

Then we need to determine the relevant control signals which are given below

Table 4 Some Control signals for the LOAD instruction

Bit in Control Memory	***Micro-operation***	***Meaning***
C0	CAR<=CAR+1	Control Address Increment
C1	CAR<=***	Control Address Redirection, depends on the position of microinstruction
C2	CAR<=0	Reset Control Address to zero position
C3	MBR<=memory	Memory Content to MBR
C4	IR<=MBR[15..8]	Copy MBR[15..8] to IR for OPCODE
C5	MAR<=MBR[7..0]	Copy MBR[7..0] to MAR for address
C6	PC<=PC+1	Increment PC for indicating position
C7	BR<=MBR	Copy MBR data to BR for buffer to ALU
C8	ACC<=0	Reset ACC register to zero
C9	ACC<=ACC+BR	Add BR to ACC
C10	MAR<=PC	Copy PC value to MAR for next address
…	……	……

Then according to the control flowchart and the table, the microprogram and control signals of the LOAD instruction is:

Table 5 Microprogram for LOAD instruction

Microprogram	***Control signals***
MBR<=memory, CAR<=CAR+1	C3, C0
IR<=MBR[15..8], CAR<=CAR+1	C4, C0
CAR<=*** (*** is determined by OPCODE)	C1
MAR<=MBR[7..0], PC<=PC+1, CAR<=CAR+1	C5, C6, C0
MBR<=memory, CAR<=CAR+1	C3, C0
BR<=MBR, ACC<=0, CAR<=CAR+1	C7, C8, C0
ACC<=ACC+BR, CAR<=CAR+1	C9, C0
MAR<=PC, CAR<=0	C10,C2

You can draw all of the control flowcharts for each instruction and determine corresponding control signals descriptions.

Before you start to design, please refer to the relevant chapters of the textbook ***<<Computer Organization and Architecture-designing for performance>>***.