CS2310: Final Assignment

Computer System Design: Gajendra-I

(submission deadline: 14th November)

You have already been introduced to the basics of a computer system and its various internal components along with their interactions. In this final assignment, you need to submit your final design, a report summarizing the design and the simulation source files/project for `CircuitVerse`. In the following, I will mainly describe the content to be put into the report. The same flow may be adapted for your final circuit build.

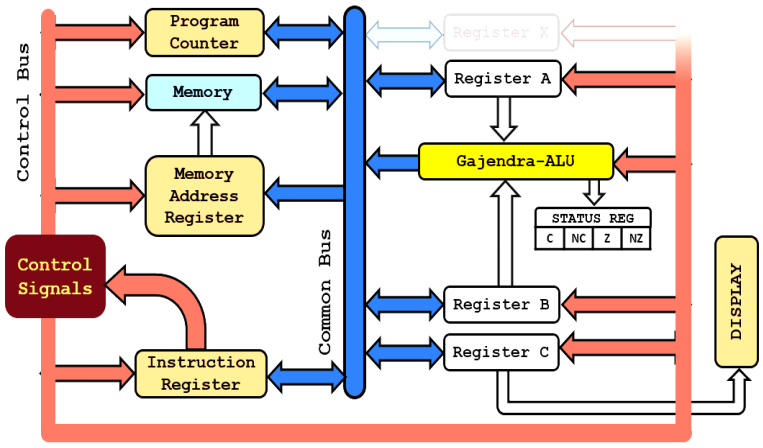
Before getting started, gloss over a sample processor manual and instruction set to get a feel about how real life datasheets look like. This manual is for the AVR processors, the same MCU used on Arduino boards. You don't have to replicate this documentation exercise by any means, this is primarily for your self-calibration. However, we need your report to be neatly structured, readable and articulate. Also, refer to the required textbook for this part, Malvino (primarily the `SAP-1` design, though our design will vary)

Overall, the design will be using 8-bit words for both data as well as instructions. The data and the program (instructions) will reside in the same memory. We will use a common bus architecture as discussed in class. The control bus will be separate. *I do not encourage you to change this unless you have an excellent grasp on the bus design (extra credit assignment) and can try out independent address and data buses for increasing the address space*.

Section 1. State the overall architecture for your CPU Core (`ARCH_GAJENDRA`)

`CircuitVerse` module: **`cpu_core_arch_gajendra`**

Start with defining the various components for your processor core. A schematic, as I have discussed in class, is shown below. You are encouraged to draw your own diagram for your design.



The system clock, memory module and the display module are not parts of the core but interfaced with the processor core. You can define how many general purpose registers you want. State how they may help to improve software flexibility. Ideally, you can use three of them, A (accumulator), B and C.

Define all the internal components and state their design along with a circuit diagram.

1. General CPU Registers (**`reg_cpu_8`**)
2. Instruction Register (**`reg_IR`**)
3. Memory Address Register (**`reg_MAR`**) - 4-bit addresses
4. Program Counter (**`counter_PC`**) - 4-bit addresses
5. Arithmetic and Logical Unit (**`ALU`**) - supports only **`ADD/SUB`**
6. Status Register (**`reg_status_4`**)**`- Z, NZ, C, NC`**

Note that **`MAR`** and **`PC`** need to deal with only 4-bit addresses. But since they are interfaced with the common 8-bit bus, make necessary arrangements for stripping off the irrelevant four bits.

The ALU for Gajendra-I design may only support unsigned addition and subtraction.

Test every component as rigorously as possible. If a particular component malfunctions the entire design will be faulty. Do not gloss over bus contention errors, if it exists there must be some sort of short circuiting - e.g., tri-state buffers are not used properly. It is your responsibility to debug the circuit. I will not debug your circuit.

Section 2. Define your Instruction Set (IS)

This does not imply simply stating the instructions you need to perform. You need to provide additional syntax details from a software programmer's perspective. One AVR example, as discussed in the class, is shown below. Look at this (interactive) for more.

ADD – Add without Carry

Description

Adds two registers without the C Flag and places the result in the destination register Rd.

Operation:

(i) (i) Rd ← Rd + Rr

	Syntax:	Operands:	Program Counter:
(i)	ADD Rd,Rr	0 ≤ d ≤ 31, 0 ≤ r ≤ 31	PC ← PC + 1

16-bit Opcode:

0000	11rd	dddd	rrrr

For your design you MUST mention which registers are impacted by the instruction, otherwise the programmer may inadvertently lose data. For this design, we intend to use 8-bit opcodes consisting of (MSB) 4-bits for specifying the operation and (LSB) 4-bits for specifying address or immediate data.

Assembly	Machine Code		Assembly	Machine Code
NOP	0000	0x0	JMP	0110 0x6
LDA	0001	0x1	JNZ	1000 0x8
STA	0010	0x2	SWAP	1001 0x9
ADD	0011	0x3	…	
SUB	0100	0x4	Maybe some MOVs	
LDI	0101	0x5	…	
OUT	0111	0x7	HALT	1111 0xF

What instructions you want to support on your processor (and their opcodes) is completely up to you. You can start with the ones we defined as a part of the class lectures, but you are free to deviate from that. At the minimum you must have load/store from address, load immediate, addition/subtraction, unconditional jump and display output (hex)

Section 3. Give a few example assembly programs implemented using your IS

I am providing some representative problems.

a. Adding two numbers and displaying the result
b. Adding and subtracting four numbers in some combination (e.g., 17-8+25-12)
c. Adding numbers from a starting address to an ending address and displaying the result. For example, add all numbers from memory location 0x9 to 0xD and display the result
d. A multiplication routine using repeated addition
e. Sorting. If you support compare and conditional jump instruction (not required in the basic design)
f. Feel free to add more …

Section 4. Microinstructions and Controller Logic Design

`CircuitVerse` module: **`cpu_timing_controller`**

Define your own control word format. The size of the control word is up to your design. If you have more registers instead of using a control bit for each IN/OUT, you may have a mux/decoder based logic to read/write from/to registers.

```
NOP:  1 << PC_out | 1 << MAR_in                  T0
      1 << PC_inc | 1 << MEM_out | 1 << IR_in    T1
      0                                          T2
      0                                          T3
      0                                          T4
```

```
LDA:  1 << PC_out | 1 << MAR_in                  T0
      1 << PC_inc | 1 << MEM_out | 1 << IR_in    T1
      1 << IR_out | 1 << MAR_in                  T2
      1 << MEM_out | 1 << REGA_in                T3
      0                                          T4
```

```
ADD:  1 << PC_out | 1 << MAR_in                  T0
      1 << PC_inc | 1 << MEM_out | 1 << IR_in    T1
      1 << IR_out | 1 << MAR_in                  T2
      1 << MEM_out | 1 << REGB_in                T3
      1 << ALU_out | 1 << REGA_in                T4
```

For each instruction define the sequence of microinstructions or control words that need to be executed in sequence. **Clearly document them for each instruction, stating the number of minimum T-states or machine cycles needed.**

Designing a hardware controller for the assignment is optional but you are expected to know how to design it.

Software based microprogrammed controller design:

- **T-states:** Use a fixed number of T-states for all instructions to keep the design simple. An internal up counter needs to be used inside the controller to indicate the step in the timing sequence. If the maximum number of T-states required is N, the internal counter can be of type mod-N (starting from zero).
- **Instruction:** The operation bits from the opcode needs to be passed from the instruction register (IR) to the controller to interpret what instruction needs to be executed.

The operation bits (e.g., 4) along with the number of bits required to encode the T-state (e.g., 3) should together form the address to "lookup" the specific control word from the control ROM. Write a Python script or a C program to automatically generate the control words for your control ROM. This automation indeed saves a lot of time.

Section 5. Connect the data/program memory and display, System Reset

Finally connect the memory module (for storing your code and data) and connect it to the processor. Put some of the sample programs that you intended to run (in Section 2) and try it out on your newly minted Gajendra-I. You can use a TTY display for your computer's display screen - see an example here.

System Reset. Have a reset button that reboots the computer so that it starts executing instructions from address 0x0. The reset input should reset the `PC` count to zero and also the internal counter within the controller.

If the programs are successfully executed giving the expected outputs, give yourself a treat :)

Submission Instructions:

a. The above report needs to be submitted in the PDF format. A great deal of importance will be given to the clarity of your documentation, neatness of your report, overall readability and presentation. Name it **report.pdf**

b. `CircuitVerse` source files for the entire project. Again, a great deal of importance will be given to the neatness of your final circuit, meaningful labels, textboxes to document things like your opcode or control word format. Pay attention to the modularity of your hardware components. Most of the components should be just a simple plug and play and some minor changes. I see many of you have made your designs public in your `CircuitVerse` account. Please make it private for now. Name the overall project **group_number_gajendraone**

c. In a text file called **aux.txt**, state references including online resources you have used. State the percentage contribution of each team member. Also mention the

roll number of students with whom you have discussed. Sharing simulation files among groups is completely prohibited.

Put all the files in a folder called **group_number_gajendracpu.zip** and upload the zipped file to Moodle. **Note, marks <u>will be deducted</u> for not following the submission naming conventions precisely.**

Additional (extra credit) extensions

applicable for both CS2300 and CS2310

1. **[Simple]** Update the ALU to support instructions like CMP (compare), SWAP, some MOV instructions, SHIFT (Left/Right) etc. A program must be written to demonstrate the capability of such instructions.

2. **[Moderate]** **Conditional Jump.** At Least **`JNZ`** - jump if not zero. Will be helpful for software looping controls. For this better to use the NZ flag to address the control ROM itself. e.g., **`NZ + 4_bit_INS + 3_bit_TSTATE.`** Note that **`NZ = 0`** or **`1`** will not impact any control word except **`JNZ iff NZ = 1. JNZ`** with **`NZ = 0`** is similar to **`NOP`** microinstructions. Only for that we are "wasting" twice the control ROM space.
   (*I understand this implementation is next to being idiotic, but better to have a working solution first before optimizing it*)

3. **[Advanced]** Attempt a separate program memory (**`PROG_MEM`** where program is stored, this can be still 16 bytes) and a data memory where data is stored (**`DATA_MEM`**, e.g., memory instructions like LDA, STA, ADD, SUB etc., will refer to this memory. You can use RAM or EEPROM to store data). Extend the bus-size to, say, 14 bits. 4 bits can be used for the instruction as usual and 10 bits can be used to address the memory.
   Note that now instead of a single memory, you have **`PROG_MEM`** and **`DATA_MEM`** - you will need to incorporate new bits in your control word, e.g., **`PROG_MEM_OUT`**, **`DATA_MEM_IN`**, **`DATA_MEM_OUT`**. **`MAR`** and **`PC`** need to be used only to control **`PROG_MEM`**. However, for instructions that need memory referencing you need to carefully orchestrate the control word to read from **`DATA_MEM`**. (*May require register indirect loading/storing instructions*)
   Write a program to sort 1024 bytes stored in `DATA_MEM`.

4. **[Advanced]** As you can appreciate, the total number of unique control words is limited, each denoting a certain state of the processor. Also, observe that the state transitions are evident and based on the T-State value and the instruction itself.

    a. Design the control logic using a Finite State Machine (Moore FSM), as an alternative to the software based controller developed in section 4 (**`cpu_timing_controller_FSM)`**.

    b. Implement this FSM using D or JK flip-flops.

    c. Benchmark your sorting program (e.g., total machine cycles spent) for **`cpu_timing_controller_FSM`** and **`cpu_timing_controller.`**

    d. Can you make an analysis of the cost-performance tradeoff between the FSM based control and software based control ROM? Cost can be approximately proportional to the number of transistors and the performance is the latency of the benchmark code.