In this exercise you are provided with an incomplete design of a datapath supporting a subset of the MIPS instructions. What you have to do is to complete the design and also design a microprogrammed control unit for it. Before we start I would like to provide you with some background required for the exercise.

**MIPS registers:**
MIPS has thirty-two 32-bit registers. Registers S0 to S7 and T0 to T9 are used for general applications. (In the table at the end of the section you can find the mapping of registers names and their numbers) Other registers have specific functions. For example:

- Register “zero” (which is in fact register number 0) always contains the value 0. (Assignments in MIPS are done by adding 0 to some register and storing the result in another register.)
- Register “ra” (return address) saves the return address of functions.
- Registers a0-a3 are used for parameter passing.
- Registers v0-v1 are used to return the value of functions.
- “fp” (frame pointer) which points to the first byte of the function activation record.
- Register “sp” points to the top of the stack at all times.

There is a difference between s0 to s7 and t0 to t9: The s-registers should be preserved during a function call meaning that if the callee is to manipulate them, it should first save a copy of them and restore it when returning the flow of control back to the caller. On the other hand, if the caller wants the value of some t-registers intact during a call, it should save them in some way before the call.

Below is a list of registers with their associated register numbers:

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Number</th>
<th>Name</th>
<th>Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Zero</td>
<td>8-15</td>
<td>T0-T7</td>
<td>28</td>
<td>GP</td>
</tr>
<tr>
<td>1</td>
<td>At</td>
<td>16-23</td>
<td>S0-S7</td>
<td>29</td>
<td>SP</td>
</tr>
<tr>
<td>2-3</td>
<td>V0-V1</td>
<td>24-25</td>
<td>T8-T9</td>
<td>30</td>
<td>FP</td>
</tr>
</tbody>
</table>
MIPS instruction formats:
The first format introduced is the R-Format which is used by register operations i.e. instructions that operate on the value of two registers saving the result in a third one.

| opcode: 6 | rs: 5 | rt: 5 | rd: 5 | shamt: 5 | func: 6 |

Here rs and rt are the source registers and rd is the destination register. Shamt specifies the amount of shift that should be applied to the result after the operation. If no shift is required it can simply be set to 0.
For most arithmetic operations in addition to opcode a function code is also required. This function code specifies the variant of the operation to be performed. For example addition and subtraction have the same opcode (0) but different function codes.

Example:
Instruction: add $t0,$s1,$s2
Machine Code:

```
100011 11110 10010 01000 00000 10000
```

Please note the usage of the “$” when referring to register.
To subtract, it suffices to change the func field to 34.

Next is the I-format used by load, store and conditional jump instructions:

| opcode: 6 | rs: 5 | rt: 5 | Offset/Displacement: 16 |

MIPS can address up to $2^{32}$ bytes of memory. LW (Load Word) and SW (Store Word) add the specified offset and the 32-bit address in rs to calculate the effective address. Then they load or store the word from or into rt.

Example:
Suppose fp contains value 3001AB08H and offset contains FDFCH as shown in the following figure:

```
000000 10001 01000 1111 1101 1111 1100
```

Then the effective address would be 3001A904H and the 32-bit value in 3001AB04H will be saved in S0.
Note: Opcode of LW is 35 and Opcode of SW is 43.
Note: MIPS is a low-endian architecture.
When using this format for conditional branches (namely beq and bne) the displacement relative to PC+4 is specified in the 16-bit long offset/displacement field. (Can you tell why PC+4 and not PC? Hint: Consult the PDP-8 architecture and see when it increments the value of PC. Also note that all MIPS instructions are 32-bits long and therefore every time an instruction is fetched, PC has to be incremented by 4.) Since instructions can only be stored at word boundaries, their addresses have always 00 as their two low-order bits. Therefore when using this format, the two low-order bits of the displacement need not be stored. This way, jumps to memory locations as far as $2^{17}$ bytes from PC+4 are made possible.

Example:
Suppose in address 30014010H the following instruction code has been saved:

```
Beq 01000 01001 0000 0000 0000 0001
```

Then if the value of S0 equals with the value of S1 the next instruction will be the one at 30014010H + 4 + 1*4 = 30014018H.

Another format is the J format which is used by branch instructions. Examples are jump which branches unconditionally and jal which saves current PC+4 into ra and jump to the specified location.

```
 Opcode: 6
 0000 0000 0000 0000 0000 0000 01
```

The address bits are shifted 2 bits to the right (why?) forming a 28 bit value. This value is then concatenated with the 4 high-order bits of PC and then a jump to the figured out location takes place.

Example:
Suppose in address 30014010H the following instruction code has been saved:

```
Opcode: 6
 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 01
```

Then the next instruction will be fetched from 0011 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0100.

Note: If higher freedom is needed “jr” can be used which jumps to an address stored in the specified register.

**The Exercise**
And now the exercise:
1. Consult the attached page and see the suggested architecture. In the diagram shown answer the following:
   - Which signal assertions, out of those shown in state 0 serve the task of incrementing PC by 4? Would this be possible without using the wire marked w?
Why? Is this step required in branch operations? (Hint: it is required. Explain why.)

- In addition to arithmetic instructions, load, store, and branch instructions need to use the ALU as well. For example, load adds rs and offset to figure out where data should be loaded/stored from/at. Design the appropriate logic for the ALUOp control wire and the ALU control box and replace the question marks with appropriate values. (hint: consider what parts of the instruction register can each of them access. + Assume that all arithmetic operations have opcode 0 and what ALU should do is determined only by the value of func i.e. it suffices to transfer func field bits to the ALU function select lines.)
- Explain the mechanism by which an arithmetic /jump /conditional branch /store /load instruction is executed. (Fully explain what happens in each state according to the values shown for the control signals.)
- What are the two shift logics required for?
- What instruction is the lengthiest? Why?

2. Write down the RTL statements for the architecture. (Use a sequence counter.)

3. Design the microprogrammed control unit for the architecture.

*Every function "execution" comprises two parts: data and the code. The code for a function is fixed and stored in a fixed position in the memory. What can differ from one execution to another, is the data. (Function parameters or local variables are two examples of what we refer to as "execution data" here.) In a recursive function, for example, every time the function calls itself a new activation record is created and the function (i.e. the fixed code) is called again with the new activation record. Activation records are kept in the stack area. When a function wants to call another function (called the caller and callee functions, respectively) it first creates the activation record by putting the parameters for the callee on the top of the stack and then calling it. When callee starts its operation, it, too, allocate some space on the stack for its local variables. These local variables are a part of the activation record, too. Therefore, as you can see, some parts of the activation record are created by the caller and some other parts by the callee. To enable a function to address the variables within the activation record, a pointer to the beginning of it is kept in a register. The function addresses its variables using this pointer as the base, adding to it an offset that identifies that specific variable. In MIPS, fp (frame pointer) is the register responsible for this. In 80x86, register BP (base pointer) does the job. As a **bonus exercise** you can suggest what other items should be kept in the activation record (in general or in MIPS) and specify why there are needed and which party (caller or the callee) puts them in the activation record.

(Note: The primary parameter passing mechanism in MIPS is not through activation records. Registers a0 to a3 are there to serve this purpose. Also registers v0 and v1 are used to return the calculated value of the callee to the caller.)