# BİL331 Bilgisayar Organizasyonu (Computer Organization)

Fall 2017

Report for final project

32-Bit Single Cycle MIPS Processor

using Verliog for implementation

Deniz Can Erdem Yılmaz

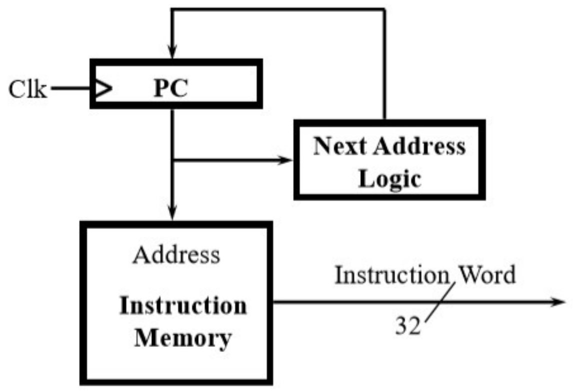
151044001

## 1. INTRODUCTION

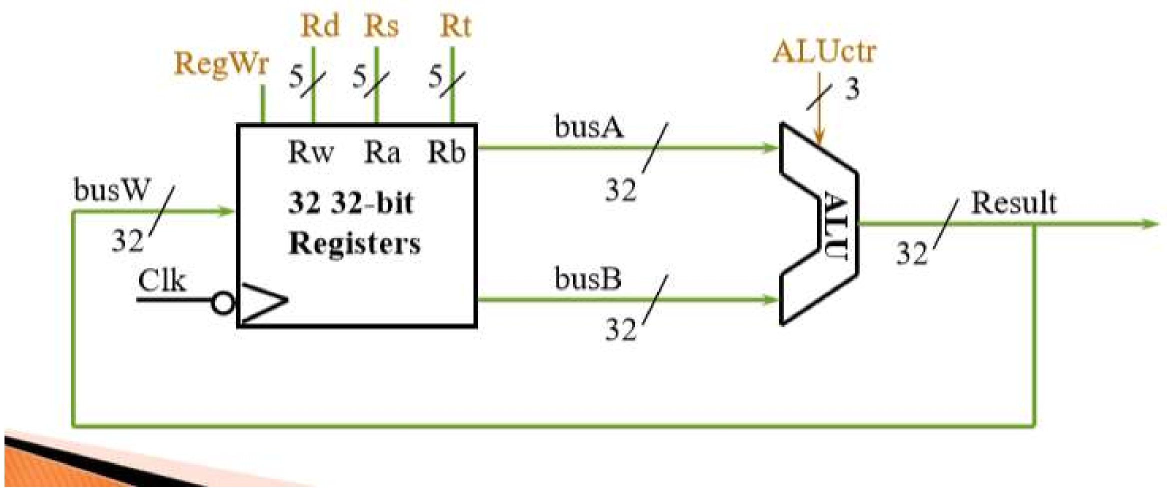
This is the report for the final project for a GTU lecture; BİL331 Bilgisayar Organizasyonu (Computer Organization), fall 2017. I will explain how and why did I do things to complete this assignment.

### 1.1 Big Picture

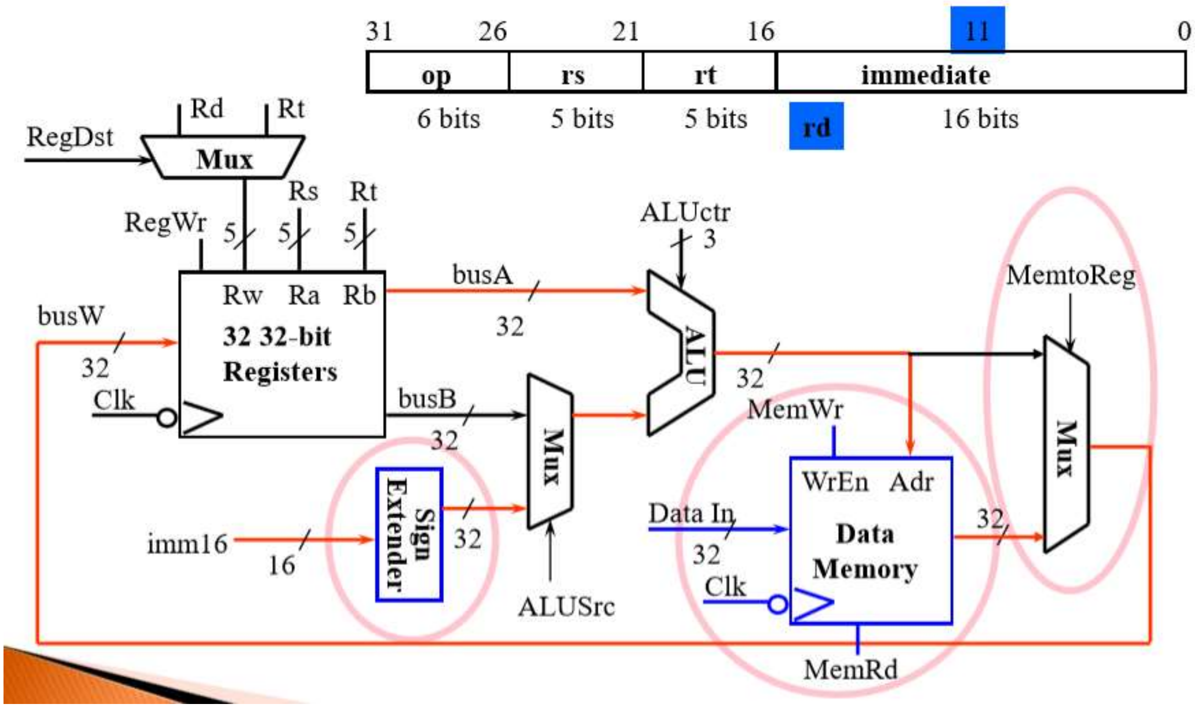
I started design with implementing parts from the single cycle data path of MIPS processor. Which can be found both in our textbook and the lecture slides. Here are the images for visual help to understand the project.



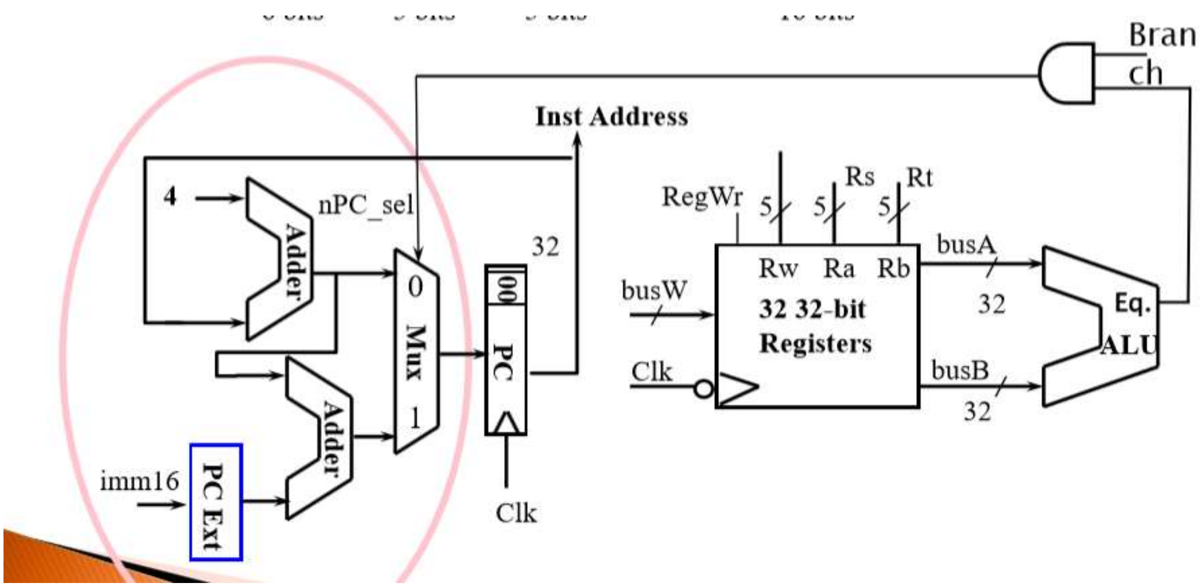
(Figure 1.1)



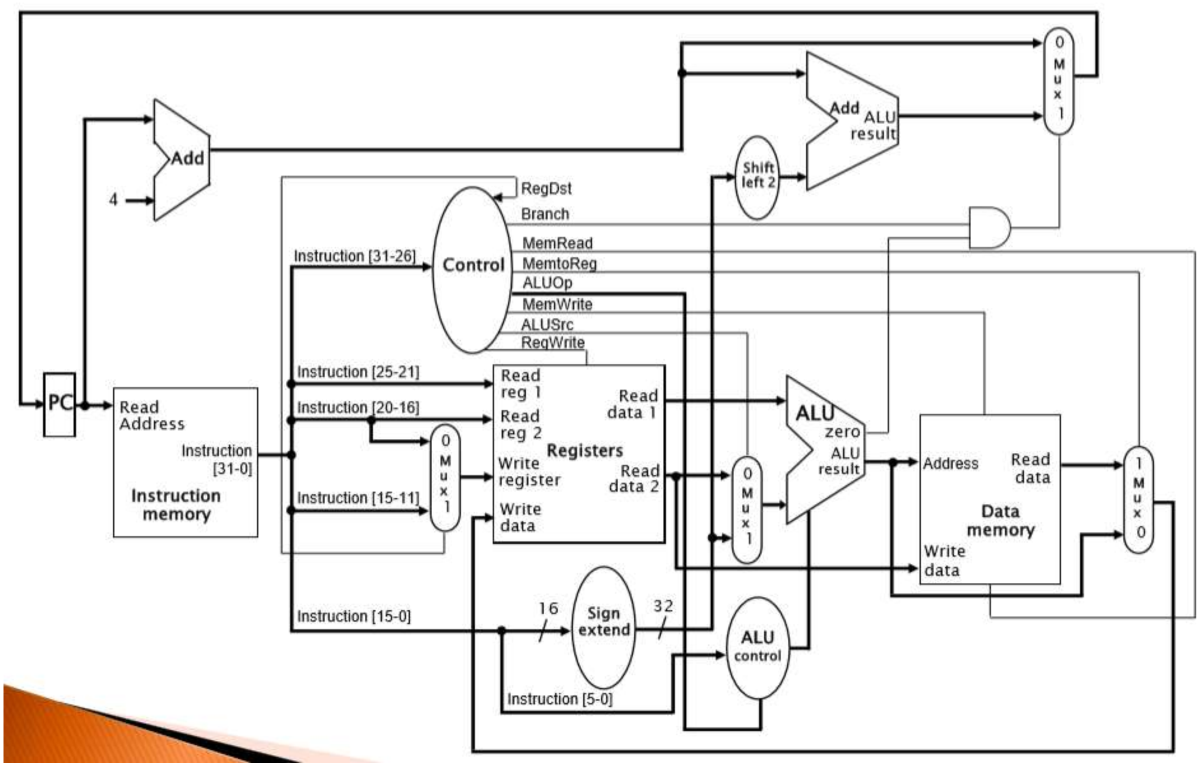
(Figure 1.2)



(Figure 1.3)

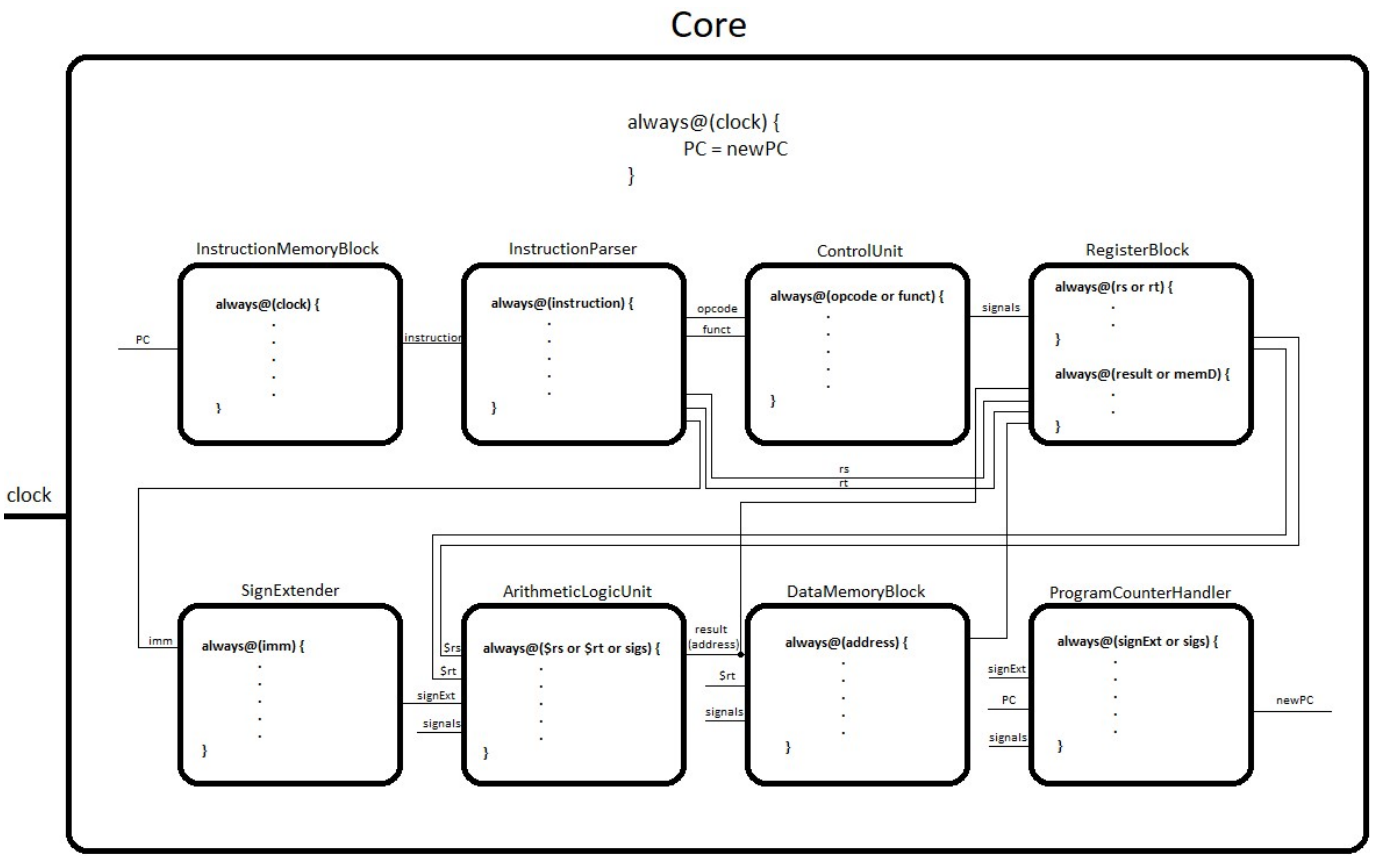


(Figure 1.4)



(Figure 1.5)

After implementing small modules, I started to combine modules in a main module named Core. Combining with the same order of the figures on top, my single cycle MIPS processor slowly begin to shape to support desired instructions. The final for (Figure 1.5) doesn't support jump (J-Type) instructions, so I additionally implemented them too. Since I am using Verilog to implement all these, next image (Figure 1.6) will be more suitable with my implementation of MIPS processor.



(Figure 1.6)

## 1.2 Lifecycle of An Instruction

The design relies on triggers established thanks to always@ blocks. Main module, Core, takes only 1 input which is the system clock. On positive edges a trigger assigns new value of program counter (*newPC*) to old value of program counter (PC).

Change of PC triggers the module *InstructionMemoryBlock*. This module takes PC as input and assigns the instruction matches with this index to the output port.

Change of instruction triggers module *InstructionParser*. This module takes instruction as input and splits instruction into the desired fields. In this stage the instruction is unknown. So, it splits all fields; *opcode*, *rs*, *rt*, *rd*, *shamt*, *funct*, *imm*, address. All these fields are output ports of this module

Change of opcode and function triggers module *ControlUnit*. This module takes opcode and function fields as inputs and produces desired control signals to operate on datapath. This is where the instruction can be identified. There are 13 different signals coming from output port of this module.

Change of *rs* and *rt* also triggers module *RegisterBlock*. This module takes signals, *rs*, *rt* and result, *memData* as input to read or write data to registers. result and *memData* will be explained in further modules. They are data candidates to be written on registers.

Change of *imm* triggers module *SignExtender*. This module takes *imm* field which is 16-bit immediate value and sign extends it to 32-bit for further usage. 32-bit version assigned to the output port.

Change of *rs* content, *rt* content or signals triggers module *ArithmeticLogicUnit* (*ALU*). This module takes 2 inputs (actually takes 3 inputs but changes 2 of them) and makes arithmetic or logical operations on them. Result will be assigned to the output port.

Change of ALU result triggers module *DataMemoryBlock*. This module takes an address (result of *ALU*), signals and a value to write if indicated by signals to write. Otherwise reads data and assigns to the output port.

Change of *signExtend* and signals triggers module *ProgramCounterHandler*. This module takes current *PC*, extended immediate field and signals to determine the next value of *PC*. It will be assigned to output port as *newPC*.

Thus, a cycle ends with an execution of instruction independently from type of instruction. When next cycle arrives (next positive edge of clock) *newPC* will be assigned to PC and all steps will be repeated.

## 1.3 Missing Parts & Bonus Parts

There are neither missing parts nor bonus parts. This is a simple single cycle data path with control unit which supports the instructions indicated with red rectangle on project definition PDF. I *hope* there is no missing or wrong parts.

## 2. Method

In order to explain how this things work all together to construct a processor I will explain each method as detailed as I can.

### 2.1 Module RegisterBlock

Inputs: *aluData*, *memData*, *pc*, *rs*, *rt*, *rd*, *regWrite*, *regDest*, *jalSignal*, *memToReg*

Outputs: *readData1*, *readData2*

Explanation: This module has an initial part which reads a .mem file (registers.mem for example) to fill up contents with register values. Also zero assigned to the first index (registers[0]) of registers as it represents the zero register which can only have 0 as value.

There are two always@ blocks which represents register read and register write separately. If *rs* (source register) or *rt* (target register) changes this block will be triggered. Contents of *rs* and *rt* will be assigned to the outputs readData1 and readData2

Second always@ block will be triggered if *aluData* (result of the ALU) or *memData* (result of the memory read stage) changes. First there will be a selection of *which* and *where*. If *regDest* signal is 1, content will be written to *rd* (destination register) else it will be written on rt. Then, if *memToReg* signal is 1, content of memory read will be written to destination, else result of the ALU will be written. Then *regWrite* signal will be controlled to see if control unit permits to write data on registers. Also, there will be an extra check for *jalSignal* to see if instruction is *jal*. If it is, then the next value of PC will be written to *ra* register.

### 2.2 Module DataMemoryBlock

Inputs: *memRead*, *writeData*, *memWrite*, *address*, *storeSig*, *loadSig*

Outputs: *readData*

Explanation: This module has an initial part which reads a .mem file (data.mem for example) to fill up contents with memory values. Also, there are constant like variables assigned to check which load or store instruction is executing to operate that way.

There is an always@ block that triggered when input *address* (result of ALU) changes. There are 2 signals named *memWrite* (permits to write data to memory) and *memRead* (permits to read data from memory). A selection will be made once always block has triggered. Signals can’t be 1 at the same time since an instruction can’t be both load and store at the same time. Either can be 1 or both can be 0. If both signals are 0 then no changes will be made on that block. If any of them is 1 then another selection will be made to see which load or store function it is, using constant like variables defined on initial block. And operation will be executed

## 2.3 Module InstructionMemoryBlock

Inputs: *instruction*

Outputs: *programCounter*

Explanation: This module has an initial part which reads a .mem file (instruction.mem for example) to fill up contents with instructions.

There is an always@ block that triggered when input *programCounter* (PC) changes. When PC changes this module will read data (instruction) written on instruction memory and assign to output *instruction*. This is the fetch part of the instruction execution cycle.

## 2.4 Module ArithmeticLogicUnit

Inputs: *jrSignal*, *data1*, *regData*, *immData*, *aluSrc*, *select*

Outputs: *result*, *zero*

Explanation: This module has an initial part which assigns constant like values to variables to compare select signal with

There is an always@ block that triggered when *data1* (content of *rs*), *data2* (content of *rt*), *select* (ALU signal to select operation) or *immData* (32-bit sign extended version of 16-bit immediate data). When triggered, this module first takes 2 of 3 input datas given by using *aluSrc* (select content of rt or immediate field) signal. If *aluSrc* is 1 data2 will be sign extended version of immediate data, else it will be content of *rt* register. Then module compares *select* signal to constant like values defined on initial block to determine which operation to execute. This ALU can operate and, or, addition, subtraction, shift, lesser data check. Assigns the result to the output port *result*. At the end of operation, also checks if result is 0. If it is output port *zero* will be 1, otherwise. This will be used on further operations (for *beq* and *bne* instructions)

## 2.5 Module ControlUnit

Inputs: *opcode*, *funct*

Outputs: *storeSig*, *loadSig*, *jSignal*, *jrSignal*, *jalSignal*, *regDest*, *branchEqual*, *branchNotEqual*, *memRead*, *memToReg*, *aluSelect*, *memWrite*, *aluSrc*, *regWrite*

Explanation: This module has an initial part which assigns constant like values to variables to forward signals to ArithmeticLogicUnit and DataMemoryBlock to understand.

There is an always@ block that triggered when *opcode* (instruction[31:26]) or *funct* (instruction[5:0]) changes. When triggered first all signals are reset to 0. Also a variable *rType* is evaluated to see if instruction is R-Typed to consider function field while assigning signals. This is the brain of the datapath. Also decode part of the instruction execution cycle. There are 14 different signal outputs. Each explained in the modules where they are used to operate. Signals produced depending on *opcode* and *funct*. Opcodes and function codes are hard-coded inside the module.

## 2.6 Module SignExtender

Inputs: *in*

Outputs: *out*

Explanation: There is an always@ block that triggered when *in* (16-bit immediate field of the instruction) changes. When triggered it will extend the 16-bit field to 32-bit and assign it to output port *out*. This module seems pretty useless right know after I learned how to use Verilog to extend or concatenate wires. But it is still the part of family.

## 2.7 Module ProgramCounterHandler

Inputs: *oldPC*, *imm*, *jumpAddress*, *jumpRegister*, *branchEqual*, *branchNotEqual*, *zero*, *jSignal*, *jrSignal*, *jalSignal*, *clock*

Outputs: *newPC*

Explanation: There is an always@ block that triggered when *clock* (system clock) has changed. When triggered it will check signals *branchEqual*, *branchNotEqual*, *zero*, *jSignal*, *jrSignal* and *jalSignal* to determine the next value of the *programCounter* (*PC*). Since instructions supplied as a file that has different instruction on each line, *PC* works as an index to that lines rather than a memory address that increases 4 each time. Each signal (except *zero* signal) represents an instruction and change PC differently from each other. *zero* signal used to check if register values are same to help instructions *bne* and *beq*, they are used along with *branchEqual* and *branchNotEqual* signals. New value of *PC* will be assigned to output port *newPC* to change *PC* and execute next instruction.

## 2.8 Module InstructionParser

Inputs: *instruction*

Outputs: *opcode*, *rs*, *rt*, *rd*, *shamt*, *funct*, *imm*, *address*

Explanation: There is an always@ block that triggered when *instruction* has changed. When triggered it will parse the current instruction given as input and assign values of fields according to it. Since the type of instruction is unknown here (it will be done when *opcode* and *funct* fields are parsed and module *ControlUnit* called) all fields are filled. Here is the meaning of fields;

*opcode* (instruction[31:26]): Each instruction (except R-Type) has different opcodes as indicators

*rs* (instruction[25:21]): Source register

*rt* (instruction[20:16]): Target register

*rd* (instruction[15:11]): Destination register

*shamt* (instruction[10:6]): Shift amount

*funct* (instruction[5:0]): Function field, selects ALU operation for R-Type instructions

*imm* (instruction[15:0]): Immediate value field

*address* (instruction[25:0]): Address to jump (for J-Type instructions)

## 2.9 Module Core

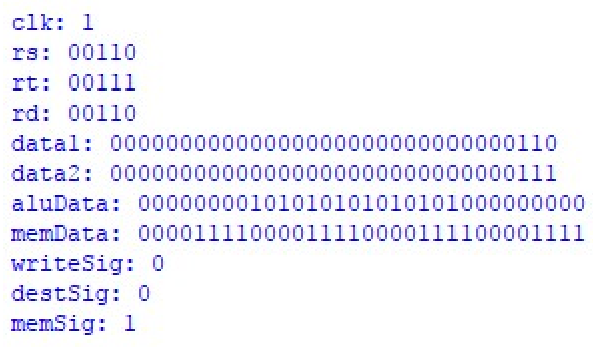
Inputs: *clock*

Outputs: -

Explanation: This is the main module that combines all the modules described above. It has an initial part to start *PC* as -1 to make it 0 when first positive *clock* edge arrives. It has an always@ block that triggers on positive edge of input *clock* (system clock). When triggered it will assign *newPC* (produced form the module ProgramCounterHandler) to the *PC* to trigger module InstructionMemoryBlock to start execution of another instruction.

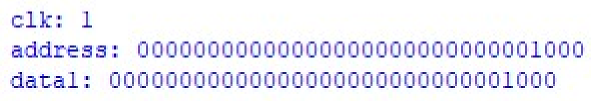
## 3. RESULT

Since there are so much options for module input and outputs there is 1 example as screen shot for each module. More variations can be tried on ModelSim to see if there is any errors or check results of specific inputs. There are explanations on both this report and project itself as comment lines for more information. Running the project with a proper instruction file will gather much more results to see and check.



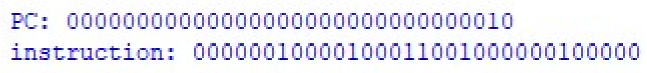
```
clk: 1
rs: 00110
rt: 00111
rd: 00110
data1: 00000000000000000000000000000110
data2: 00000000000000000000000000000111
aluData: 00000000101010101010101000000000
memData: 00001111000011110000111100001111
writeSig: 0
destSig: 0
memSig: 1
```

(Figure 3.1: RegisterBlock_TestBench result)



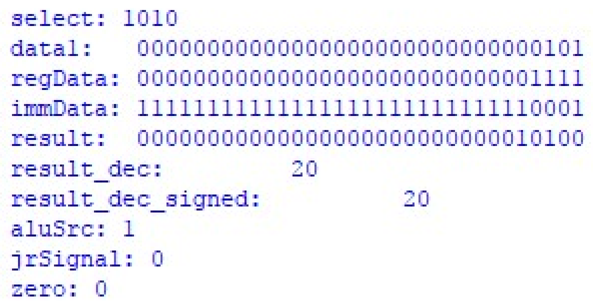
```
clk: 1
address: 00000000000000000000000000001000
data1: 00000000000000000000000000001000
```

(Figure 3.2: DataMemoryBlock_TestBench result)



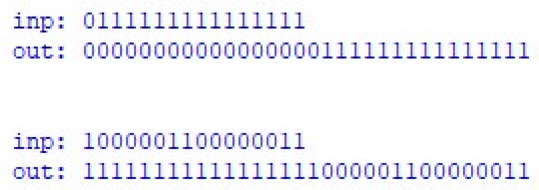
```
PC: 00000000000000000000000000000010
instruction: 00000010000100011001000000100000
```

(Figure 3.3: InstructionMemoryBlock_TestBench result)



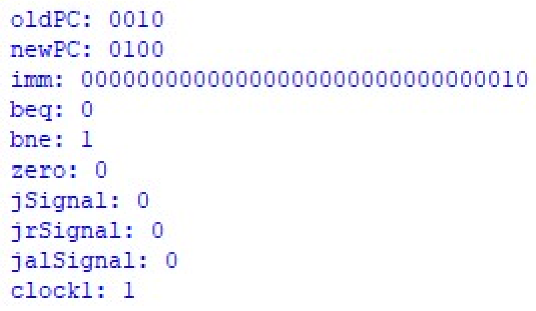
```
select: 1010
data1:   00000000000000000000000000000101
regData: 00000000000000000000000000001111
immData: 11111111111111111111111111110001
result:  00000000000000000000000000010100
result_dec:          20
result_dec_signed:          20
aluSrc: 1
jrSignal: 0
zero: 0
```

(Figure 3.4: ArithmeticLogicUnit_TestBench result)



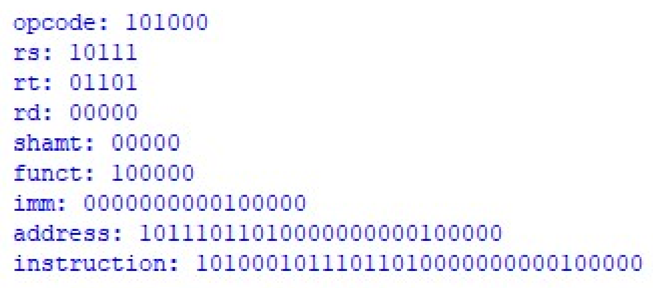
```
inp: 0111111111111111
out: 00000000000000000111111111111111


inp: 1000001100000011
out: 11111111111111111000001100000011
```

(Figure 3.5: SignExtender_TestBench result)



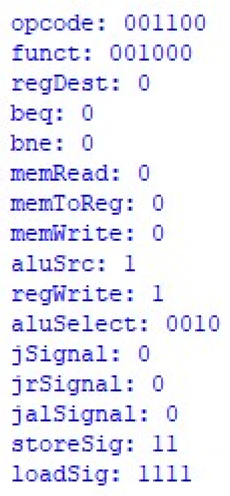
```
oldPC: 0010
newPC: 0100
imm: 00000000000000000000000000000010
beq: 0
bne: 1
zero: 0
jSignal: 0
jrSignal: 0
jalSignal: 0
clock1: 1
```

(Figure 3.6: ProgramCounterHandler_TestBench result)



```
opcode: 101000
rs: 10111
rt: 01101
rd: 00000
shamt: 00000
funct: 100000
imm: 0000000000100000
address: 10111011010000000000100000
instruction: 10100010111011010000000000100000
```

(Figure 3.7: InstructionParser_TestBench result)



```
opcode: 001100
funct: 001000
regDest: 0
beq: 0
bne: 0
memRead: 0
memToReg: 0
memWrite: 0
aluSrc: 1
regWrite: 1
aluSelect: 0010
jSignal: 0
jrSignal: 0
jalSignal: 0
storeSig: 11
loadSig: 1111
```

(Figure 3.8: ControlUnit_TestBench result)