ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 2 REPORT

CRN : 21334 & 21335

LECTURER : Gökhan İnce & Deniz Turgay Altılar

GROUP MEMBERS:

150220098 : AHMET BARIŞ ATLI

150220716 : ZEKERİYA ATBAŞ

SPRING 2024

Contents

1 INTRODUCTION [10 points]

The aim of this project is implementing a CPU System for executing programs. A CPU consists of 3 parts which are memory, control unit and an arithmetic logic unit system (ALU). The ALU system is implemented in the project 1 and the memory structure is given. Control unit needs to be implemented for the CPUSystem to be complete. For the most part of this project, a control unit will be implemented.

Every part of the experiment (implementation, testing, troubleshooting, report) are done by both Zekeriya and Barış together.

2 MATERIALS AND METHODS [40 points]

In our implementation, control unit is not implemented as a seperate module but rather together with the ALU system. It is easier to connect the control signals in the code implementation that way. Memory is already included in the ALU system, though the contents of the memory is changed for certain tests on our module. Since memory words are 8 bits long, two memory words are needed for filling up the 16-bit registers in our system. Also the values in the registers can be stored in 2 consecutive memory words. In other words, every memory word contains half of the information. For that reason, when increment or decrement operation in the address registers need to be done, it is done twice for referencing to a correct value.

2.1 Assumptions

For the ambiguity that is caused in the given project sheets, following assumptions are made and our implementation is intended to work that way.

- All of the operations are designed to work with 16 bits. In theory, any of the registers in ARF can take 16 bit values and the memory that system works with is said to have 256 memory words at most, the registers can point to the memory address that the memory can not reach. We assume that larger memory sizes can be used in the tests. For that reason, implementation is done as all of the bits of address registers are used.

- When the reset signal is sent (0), the current operation is halted. It does not matter which stage the program is on, when the reset signal comes to the system all general purpose registers and PC is cleared by manually manipulating the inside values. SP is set to the bottom value of a 8 bit memory (0000000011111111) when reset signal is 0. Also, reset signal resets the SC value to 0 for starting everything all over again.

- All the registers and modules that are not used is disabled at both within the T0 and T2. Register files are disabled by setting all the RegSel values to 1's to disable them. Mem CS and IR write values are also disabled in these cycles, if not in use. This prevents the unwanted data transfers from happening during an instruction.

- For INC and DEC instructions, when the increment or decrement operation is done it is done only once, regardless of the register type. If for example, AR needs to be incremented, AR $\leftarrow$ AR + 1 operation will be done. Though, PC and AR always need to end up with a even value and SP needs to end up with an odd value. Incrementing a value only once messes with the reading operation from the memory. It is assumed that instructions will be given once or twice by the programmer, depending on the destination register.

- "VALUE", "IMMEDIATE" and "OFFSET" values in the project sheet are always assumed to be the address bits of the IR, which are the lowest 8 bits.

- For the instructions without the address reference, required timing signals can change depending on the certain conditions. The reason being is destination and source registers can be any register in address register file (ARF) or one of the Rx registers in register file (RF) depending on its relative code. This creates different situations for the instruction because getting or loading data to ARF and RF implementations are different. Identifying where the source register is vital for this kind of operations. Note that the locations of source and destination registers can be determined by their relative bits.

DSTREG/SREG1/SREG2	**Location of the register**
0xx	ARF
1xx	RF

Table 1: Determining destination and source register locations

- For ALU operations, destination and source registers place really matters because depending on the location of the SREGs, the implementation and required timing signals of the instruction change. One of the reasons for this situation is ARF does not have a direct connection to ALU whereas RF does. If SREG's location is in the ARF, it must be loaded into one of the scratch registers in the RF first in order to access the ALU. If SREG is one of the registers in RF it does not need such thing. A value in the general purpose register can still be loaded into a scratch register using one timing signal in order to synchronize with other side of the operation, the situation when SREG is in the ARF. In our implementation, we considered

executing the instructions as fast as possible. For that reason, when SREGs are in RF, there is only one timing signal is used.

- ALU flags are always set to zero by default for all address reference instructions (for example, PC + VALUE operation for the branches or AR + OFFSET operation for STRIM will not change the ALU flags.). For other operations, especially the ALU operations, flags are checked with the S bits. It does not matter if the operation has to change flags or not, flag control signal is not hardcoded into implementation. The assumption is, when the coming instruction is defined to change the flags, the S value will always be 1 by the programmer. For example, it is assumed that when ADDS or any shift function is called, the S value can never come as 0 by the programmer. Also when the ADD function is called, which does not change the flags, the coming S bit never will be set to 1 by the programmer. In any way, the S bit is the one determining the flag write signal.

- MOV instructions are assumed to be address reference instructions rather than non-address reference instruction. In the project sheet, it is said that destination registers will be used but destination registers are a thing of non-addresss reference instruction. And it is impossible to load immediate value with this instruction set because address reference is not present. Treating MOVL and MOVS as address reference instructions and assuming destination registers only refers to Rx registers in RF, this issue is solved.

- AR and PC values are always preserved at the end of the instruction, unless stated otherwise. For loading a value to the memory whose address is referenced by these two registers, AR and PC needs to be incremented for loading high and low parts. Then these values are decremented while loading the second set of bits onto the memory.

2.2 Sequence Counter

In our implementation, a 3 bit sequence counter register (SC) is created and timing signals (Tx) are obtained from decoding the SC. Timing signals are used for the purpose of controlling which operation the system is on. With every operation done, the SC is updated. If an instruction is not done yet and cycle continues, SC is incremented, thus changing the Tx value to pass to the next operation in the cycle. If the instruction is done, SC is set to be zero. It is not necessary to reset the scratch registers every clock cycle since scratch registers are already used for temporary operations and meant to change constantly.

2.3 Fetching

In the fetch cycle, the instruction register (IR) gets the value from the memory. Since memory words are 8 bit long and instruction register is 16 bit long, two clock cycles are needed for filling up the IR and start the decoding phase. PC value needs to be incremented two times for skipping to the next operation because of the reasons that are discussed above. To do this, the FunSel of address register file (ARF) is selected to 001 (increment) and kept that way for both of T0 and T1 cycles. There is no need to send the PC value to AR value unlike basic computer because in this case all registers in the ARF (PC, AR and SP) are directly connected to the memory address and can be used for that purpose. Also, doing so will waste an additional clock cycle.

2.4 Decoding

After the load operation to the IR is complete (fetching cycle), instructions will be executed and these instructions are chosen from OPCODE portion of the IR. There are two types of instruction formats and these formats are decided according to definitions of the operations. For instructions without the address reference, even the required clock cycles can change based on which source registers are chosen for the instruction.

2.5 Execution

Execution cycle is the main part of the instruction cycle. Necessary wirings and control signals are given to the ALU system. There are thirty four operations that CPU system is executing however some of the operations are basically the same apart from few small changes. As a result of that, implementation cost is highly reduced. The brief explanation of the implemented operations are given below.

2.5.1 Branch instructions (BRA, BNE, BEQ)

Branch instructions are used for adding the program counter (PC) value to the current "VALUE" and loading the result to the PC again. This way, program can branch ta an adrress relative to the PC value with given "VALUE". It is assumed that "VALUE" in the project file is the lowest 8 bits in the instruction register (IR).

The difference between 3 of the branch operations are while BRA is unconditional branch, BNE and BEQ operates depending on the zero flag of the ALU. BNE branch is done if the Z flag is zero and BEQ branch is done if the Z flag is one. These instructions take 3 timing signals.

Instruction details for branches :

- T_2 - S1 $\leftarrow$ IR[7:0]
- T_3 - S2 $\leftarrow$ PC
- T_4 - PC $\leftarrow$ S1 + S2 (ALU FunSel = 10100)

For BNE and BEQ, condition of the zero flag is checked before operations. If the conditions does not met, system looks for the next instruction.

2.5.2 Stack instructions (POP, PSH)

Stack is controlled by stack pointer (SP) register. Initially, SP is set to the bottom of the 8-bit memory (0000000011111111). If SP shows the bottom of the memory like the initial condition, stack is empty. When the push (PSH) instruction is sent to the system, it loads the value of the chosen general purpose register (R1, R2, R3, R4) to the stack. Since values in the general purpose registers are 16 bit and memory words are 8 bit, value is loaded in 2 time signals. Little endian order of memory is protected and since SP is decremented when push operation is done, higher bits are pushed first to the memory. With that manner, little endian order is not violated. Also, SP is decremented twice. Entire push operation takes 2 timing signals.

However for the pop (POP) instruction, 3 timing signals are used. POP instruction is more or less the reverse of the PSH istruction, except the memory part that SP points is considered empty. For popping a value from the stack, SP needs to be incremented first and remain unchanged when popping the highest bits. This requires an additional cycle since one of the incrementing and popping operations have to be done seperately.

Instruction details for PSH :

- T_2 - M[SP] $\leftarrow$ Rx[15:8], SP $\leftarrow$ SP - 1 (ALU FunSel = 10000, MuxCSel = 1)
- T_3 - M[SP] $\leftarrow$ Rx[7:0], SP $\leftarrow$ SP - 1 (ALU FunSel =10000, MuxCSel = 0)

Instruction details for POP :

- T_2 - SP $\leftarrow$ SP + 1
- T_3 - Rx[7:0] $\leftarrow$ M[SP], SP $\leftarrow$ SP + 1
- T_4 - Rx[15:8] $\leftarrow$ M[SP]

Rx is determined by register select bits of the instruction register. (IR[9:8])

2.5.3 Increment, Decrement instructions (INC, DEC)

This set of instructions are different type of instruction from the above ones. These instructions does not have a address reference. Instead, parts of the instruction register is used for flag enable bit (S) and choosing the destination and source registers. (DSTREG and SREGs).

This set of increment, decrement instructions first takes the value from the source register and loads into the destination register. Then destination register is incremented or decremented depending on the instruction. Increment and decrement operations are done within the destination register and after the loading operation. In our assumption, source registers will not change and final value will be only contained in the destination register. Both instructions take 2 timing signals.

Implementation details for both INC and DEC :

- T_2 - DSTREG $\leftarrow$ SREG1
- T_3 - if INC DSTREG $\leftarrow$ DSTREG + 1 if DEC DSTREG $\leftarrow$ DSTREG - 1 (ALU FunSel = 10000)

Direct connection from ARF to RF and vice versa is possible and scratch registers in the RF are not used so the operation does not change depending on the SREG location.

2.5.4 ALU instructions with 1 source register (LSL, LSR, ASR, CSR, CSL, NOT, MOVS)

ALU instructions takes a value from the SREG1 (determined by IROut[5:3]), then it is sent to ALU and based on the instruction, FunSel of ALU is manipulated and result is loaded into DSTREG. The location of the SREG can be checked using the most significant bit it is determined with (IROut[5]). When this bit is zero, SREG is in ARF and when it is one, SREG is in RF.

Implementation details for ALU instructions with 1 source register (IROut[5] = 0 case) :

- T_2 - S1 $\leftarrow$ SREG1
- T_3 - DSTREG $\leftarrow$ LSL/LSR/ASR/CSR/CSL/NOT/ S1 (ALU FunSel = 11011(LSL), 11100(LSR), 11101(ASR), 11110(CSR), 11111(CSL), 10010(NOT), 10000(MOVS))

Implementation details for ALU instructions with 1 source register (IROut[5] = 1 case) :

- T_2 - DSTREG $\leftarrow$ LSL/LSR/ASR/CSR/CSL/NOT/ SREG1 (ALU FunSel = 11011(LSL), 11100(LSR), 11101(ASR), 11110(CSR), 11111(CSL), 10010(NOT), 10000(MOVS))

2.5.5 ALU instructions with 2 source registers (AND, ORR, XOR, NAND, ADD, ADC, SUB, ADDS, SUBS, ANDS, ORRS, XORS)

The idea for ALU instructions with 2 source registers are similar to the 1 source registers, with the addition of one more source register (SREG2). For correct implementation, location of the SREG2 is also determined with its recognizing bits (IROut[2]). These instructions take 2,3, or 4 clock signals depending on the locations of the source registers.

Implementation details for ALU instructions with 2 source registers (IROut[2] = 0 AND IROut[5] = 0 case) :

- T_2 - S1 $\leftarrow$ SREG1
- T_3 - S2 $\leftarrow$ SREG2
- T_4 - DSTREG $\leftarrow$ S1 AND/OR/XOR/NAND/+/+ Carry +/- S2 (ALU FunSel = 10111(AND, ANDS), 11000(OR, ORRS), 11001(NAND), 10100(ADD, ADDS), 10101(ADC), 10110(SUB, SUBS))

Implementation details for ALU instructions with 2 source registers (IROut[2] = 0 AND IROut[5] = 1 case) :

- T_2 - S1 $\leftarrow$ SREG2
- T_3 - DSTREG $\leftarrow$ SREG1 AND/OR/XOR/NAND/+/+ Carry +/- S1 (ALU FunSel = 10111(AND, ANDS), 11000(OR, ORRS), 11001(NAND), 10100(ADD, ADDS), 10101(ADC), 10110(SUB, SUBS))

Implementation details for ALU instructions with 2 source registers (IROut[2] = 1 AND IROut[5] = 0 case) :

- T_2 - S1 $\leftarrow$ SREG1
- T_3 - DSTREG $\leftarrow$ S1 AND/OR/XOR/NAND/+/+ Carry +/- SREG2 (ALU FunSel = 10111(AND, ANDS), 11000(OR, ORRS), 11001(NAND), 10100(ADD, ADDS), 10101(ADC), 10110(SUB, SUBS))

Implementation details for ALU instructions with 2 source registers (IROut[2] = 1 AND IROut[5] = 1 case) :

- T_2 - DSTREG $\leftarrow$ SREG1 AND/OR/XOR/NAND/+/+ Carry +/- SREG2 (ALU FunSel = 10111(AND, ANDS), 11000(OR, ORRS), 11001(NAND), 10100(ADD, ADDS), 10101(ADC), 10110(SUB, SUBS))

2.5.6 MOV instructions (MOVH, MOVL)

These instructions take one timing signal and MOVH moves "IMMEDIATE" to the higher part and MOVL loads into the lower part.

Implementation details for MOVH instruction :

- T_2 - Rx[15:8] $\leftarrow$ IROut[7:0]

Implementation details for MOVL instruction :

- T_2 - Rx[7:0] $\leftarrow$ IROut[7:0]

2.5.7 Load instruction (LDR)

This instruction loads the value in the memory address that is referenced by the AR to the chosen Rx. Instruction is done with 16 bits. AR needs to be incremented for loading 2 seperate words into the Rx but AR is decremented back to its original position. AR stays the same as the beginning after the instruction is done. This instruction takes 2 timing signals.

Implementation details for LDR instruction :

- T_2 - Rx[7:0] $\leftarrow$ M[AR], AR $\leftarrow$ AR + 1
- T_3 - Rx[15:8] $\leftarrow$ M[AR], AR $\leftarrow$ AR - 1

2.5.8 Store instruction (STR)

This instruction stores the value in the chosen Rx to the memory address that is referenced by the AR. Instruction is done with 16 bits. AR needs to be incremented for storing 2 seperate words in the memory but AR is decremented back to its original position. AR stays the same as the beginning after the instruction is done. This instruction takes 2 timing signals.

Implementation details for STR instruction :

- T_2 - M[AR] $\leftarrow$ Rx[7:0], AR $\leftarrow$ AR + 1 (ALU FunSel = 10000, MuxCSel = 0)
- T_3 - M[AR] $\leftarrow$ Rx[15:8], AR $\leftarrow$ AR - 1 (ALU FunSel = 10000, MuxCSel = 1)

2.5.9 Branch instruction with storing current program (BX)

This instruction is a type of branch instruction that pushes the current position of the program (PC) into the stack. Then it loads the value of chosen register Rx into the PC and by doing that, program is now branched to new address. BX directly branches to the

loading address unlike other branch operations. The desired branching address simply can be loaded to one of the Rx registers beforehand, then this address can be taken with this instruction to branch to desired place in the program. Other branch operations are only performing relative branching to the current PC value, which makes it harder to branch the desired address. Since pushing the current value of PC is needed, push operation is also implemented inside this instruction. Details of the push/pop operations are discussed in their relative headlines. This instruction takes four timing signals.

Implementation details for BX instruction :

- T_2 - S1 $\leftarrow$ PC (ALU FunSel = 10000)
- T_3 - M[SP] $\leftarrow$ PC[15:8], SP $\leftarrow$ SP - 1 (ALU FunSel = 10000, MuxCSel = 1)
- T_4 - M[SP] $\leftarrow$ PC[7:0], SP $\leftarrow$ SP - 1 (ALU FunSel = 10000, MuxCSel = 0)
- T_5 - PC $\leftarrow$ Rx, SP $\leftarrow$ SP - 1 (Mem CS = 1)

2.5.10 Branch to the last instruction in stack (BL)

This instruction is for branching the program back to its previous position with popping the address from stack. Details of the pop instruction is discussed in its relative headlines. The value in the memory cannot be loaded to the PC directly, since when getting value from the memory, SP needs to be incremented and both of these registers are in the same module ARF. Load and increment operations in a single module is not possible. For that reason, the value in the memory bits are stored in one of the scratch registers first. After reading of the value is done, the value in the scratch register is loaded into PC. This instruction takes four timing signals.

Implementation details for BL instruction :

- T_2 - SP $\leftarrow$ SP + 1
- T_3 - S1[7:0] $\leftarrow$ M[SP], SP $\leftarrow$ SP + 1
- T_4 - S1[15:8] $\leftarrow$ M[SP]
- T_5 - PC $\leftarrow$ S1 (ALU FunSel = 10000)

2.5.11 Load immediate instruction (LDRIM)

This instruction takes the immediate value from the IR and loads it into the chosen register Rx. It is only loading immediate value to the low bits of the Rx. This instruction takes one timing signal.

Implementation details for LDRIM instruction :

- T_2 - Rx[7:0] $\leftarrow$ IR[7:0]

2.5.12 Store immediate instruction (STRIM)

This instruction stores the current value in the chosen register Rx to a memory place relative to the AR. This offset is determined by the IR address bits (IR[7:0]). Note that current AR value needs to be preserved and loaded back into AR after the instruction is done because AR will change its value in the process. Current AR and offset values are loaded to the scratch registers, then gets added and loaded to AR. Then Rx is loaded to this new address of AR+offset. Lastly, the preserved value of AR is loaded back to AR from the scratch register. This instruction takes six timing signals which makes STRIM operation the slowest operation in this system.

Implementation details for STRIM instruction :

- T_2 - S1 $\leftarrow$ AR
- T_3 - S2 $\leftarrow$ IR[7:0]
- T_4 - AR $\leftarrow$ S1 + S2 (ALU FunSel = 10100)
- T_5 - M[AR] $\leftarrow$ Rx[7:0], AR $\leftarrow$ AR + 1 (ALU FunSel = 10000, MuxCSel = 0)
- T_6 - M[AR] $\leftarrow$ Rx[15:8],AR AR $\leftarrow$ AR - 1 (ALU FunSel = 10000, MuxCSel = 1)
- T_7 - AR $\leftarrow$ S1

3 RESULTS [15 points]

For getting the results and showing the CPUsystem is working accurately, we implemented the given program in the project sheet. The program is written in the memory (RAM.mem). This program adds values written to the certain memory part and prints their result back to the memory, to a place below the values added. The details of the program is discussed in the discussion part.

Adding 10 numbers with the simulation. (16+17+18+19+20+21+22+23+24+25 = 205) Note that all numbers are 8 bit numbers.

```
Output Values:
T:   1
Address Register File: PC:    72, AR:   196, SP:   255
Instruction Register : 4b00
Register File Registers: R1: 65535, R2:   205, R3:    25, R4:   205
Register File Scratch Registers: S1:    12, S2:    68, S3:     0, S4:     0
ALU Flags: Z: 1, N: x, C: 0, O: x
ALU Result: ALUOut:   205
Control Signals: ARF_RegSel: 011, RF_RegSel: 1111, RF_ScrSel: 1111, Instruction: 010010
Final value in the address:   0 205
```

Figure 1: Obtained results for adding numbers 16 to 25

Adding 3 numbers with the simulation (437+620+927 = 1984). Note that all numbers are 16 bit numbers.

```
Output Values:
T:   1
Address Register File: PC:    72, AR:   196, SP:   255
Instruction Register : 4b00
Register File Registers: R1: 65535, R2:  1984, R3:     0, R4:  1984
Register File Scratch Registers: S1:    12, S2:    68, S3:     0, S4:     0
ALU Flags: Z: 1, N: x, C: 0, O: x
ALU Result: ALUOut:  1984
Control Signals: ARF_RegSel: 011, RF_RegSel: 1111, RF_ScrSel: 1111, Instruction: 010010
Final value in the address:   7 192
```

Figure 2: Obtained results for adding numbers 437+620+927

4 DISCUSSION [25 points]

The program we used to test our CPU System module is taken from the project sheet. However, since the program in the project sheet is outdated, some operations did not match with what we have. For example, for BRA and BNE to work on this program, they need to get the value directly from the IR however our implementation of branch operations are branching relatively to the PC, not directly from the IR. The same program is implemented for our CPU with a bit different approach.

- **BRA 0x26** Since PC value is 2 and adding 26 will branch the program to the 28
- **MOVH R1 0x00**
- **MOVL R1 0x09** The number 0x09 is arbitrary, R1 is used for how many iterations will be done, thus how many numbers will be added. In our program, it adds R1 + 1 many of numbers. In this case, for adding 10 numbers, R1 is set to 9.
- **MOVH R2 0x00**
- **MOVL R2 0x00** R2 is used for storing the total

- **MOVH R3 0x00**
- **MOVL R3 0xB0** R3 first stores the address of the numbers that will added, then it is used for reading those numbers.
- **MOVS AR, R3** Moving the address from R3 to AR. R3 will be used for reading the numbers from now on.
- **MOVH R4 0x00**
- **MOVL R4 0x37** R4 is used for branching to the address LABEL, details will be discussed.
- LABEL (0x38): **LDR R3** The operation is in a loop between LABEL and BNE instruction. This loop will continue until BNE instruction does not branch to LABEL anymore. (when the Z flag of ALU is 1)
- **ADD R2, R2, R3** Current number and the total value is added and loaded into total.
- **INC AR, AR**
- **INC AR, AR** AR is incremented twice for avoiding corruption of memory reading. It goes to the next number to read and process at the next loop.
- **DEC R1, R1** 1 iteration is done so the R1 value is decremented
- **BNE 0x0C** Adding the current value and 0x0C address then branching to the result. That result is TEMP. The data in TEMP is used for branching to a direct value.
- **STR R2** After all the iterations are done and BNE does not branch to the LABEL anymore, the program is done. This instruction stores the result to the memory which is pointed by AR. AR points to the memory location right after the added values.
- **LDR R4** This instruction is not necessary for the program to work, it is used for showing memory is working and R4 can take the value which is written to the memory afterwards. R4 and R2 will be the final result.
- TEMP (0x50): **INC PC, R4** A temp address relative to the address which BNE instruction is present. R4 stores the LABEL address minus 1. R4 is loaded to PC and it is incremented by 1, Thus, branching PC value to the LABEL by using 2

instructions. This is a trick for branching to the desired value while having relative branches.

The reason why MOVS is not used is because we did not want the program to change the flags and only other option for loading a value from Rx to PC is by INC and DEC instructions. DEC instruction can also be used for that case (R4 would be 0x39). Also, BX provides direct branching but it stores current PC value in the stack. The current value of PC when branching to another place is useless in our implementation.

Note that our CPU implementation does not have an halt instruction, so after the instructions are done the CPU continues to read data from the memory and messes with the values. The results posted are from when the instruction is finished its job.

5 CONCLUSION [10 points]

As a final note, using Vivado, we have implemented a CPU system that is capable of running programs like a real computer. The number one difficulty we faced was ambiguity of the project sheet. When designing a CPU, the designer should precisely know what and how to implement an instruction. If there is an ambiguity in the instructions, it is hard to decide how to implement certain instructions. We asked our questions to assistants and depending on the answers we got, we did some assumptions. Our implementation is working based on the assumptions that are discussed in the methods part. We saw that registers and other logical systems are constructed from smaller and more basic parts with the project 1. In this project, we saw how small things can come together and form abstract elements that can work as a computer.

REFERENCES