FINAL PROJECT VHDL

Amit Nagar- Halevy and Tal Kapelnik
204210306 204117089

BGU
COMPUTER ENGEENIRING

Contents

Assignment description:

The aim of this laboratory is to design a simple MIPS compatible CPU. The CPU uses's a Single Cycle MIPS architecture and must be capable of performing instructions from MIPS instruction set. The design is executed on the Altera Board. The MIPS architecture is Harvard architecture in order to increase throughput and simplify the logic.
The architecture includes a MIPS ISA compatible CPU with data and program memory for hosting data and code. The block diagram of the architecture is given in Figure 1. The CPU have a standard MIPS register file. The top level and the MIPS core is structural. The design compiled and loaded to the Altera board for testing. A single clock (CLK) of 24MHZ is used in the design.

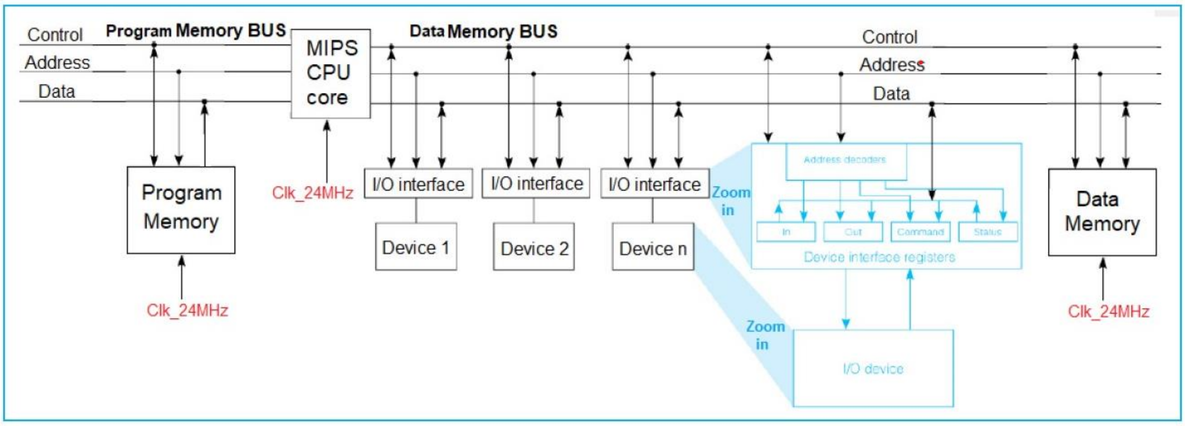
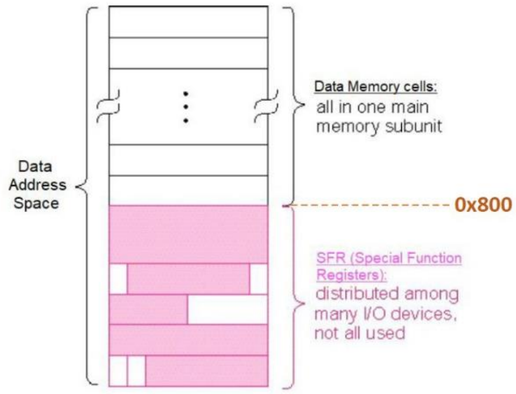


Figure 1 : System architecture



MEMORY Mapped I/O addresses:

PORT_LEDG[7-0] 0x800 - LSB byte (Output Mode)

PORT_LEDR[7-0] 0x804 - LSB byte (Output Mode)

PORT_HEX0[7-0] 0x808 - LSB byte (Output Mode)

PORT_HEX1[7-0] 0x80C - LSB byte (Output Mode)

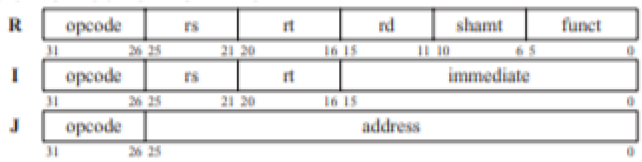
PORT_HEX2[7-0] 0x810 - LSB byte (Output Mode)

PORT_HEX3[7-0] 0x814 - LSB byte (Output Mode)

PORT_SW[7-0] 0x818 - LSB byte (Input Mode)

Supported Operation code formats:

Rtype Itype and Jtype as follow -



The supported set of opcodes in our version of single cycle Mips:

Or, ori, and, andi, xor, xori, add, addi, addu,

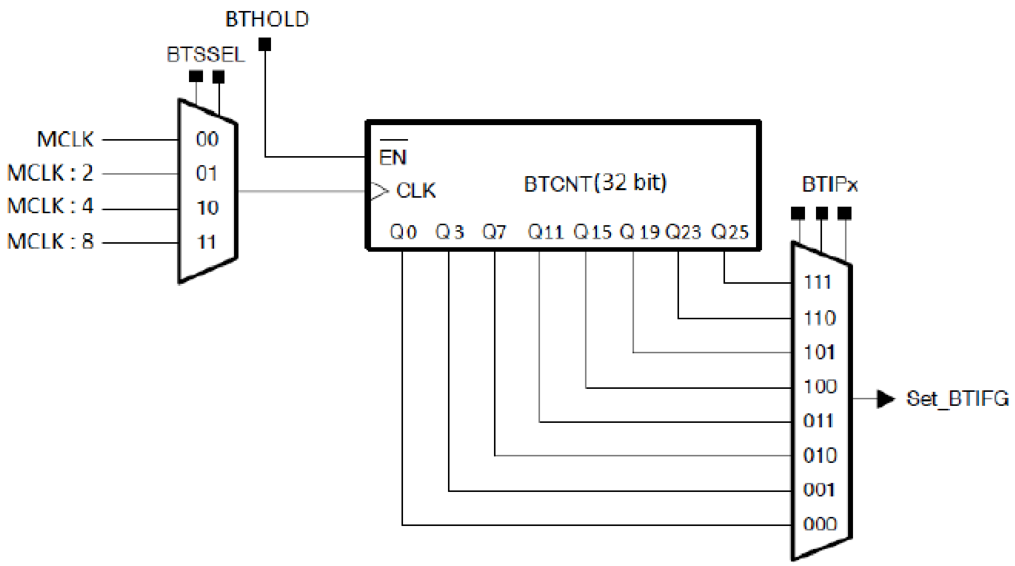
Lw, sw , slt, sll, srl , beq, bne, sub, j, jr, jal.

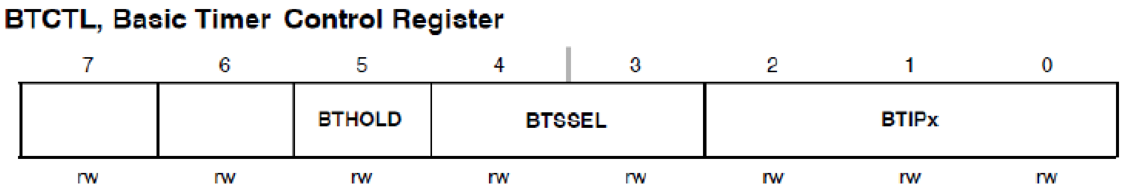
Added Peripherals:

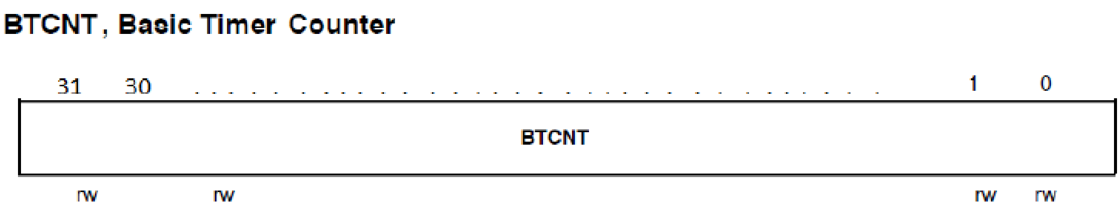
1. Key[3-0]

2. Basic Timer:

A basic timer,it can send an interrupt every time divided as follows:





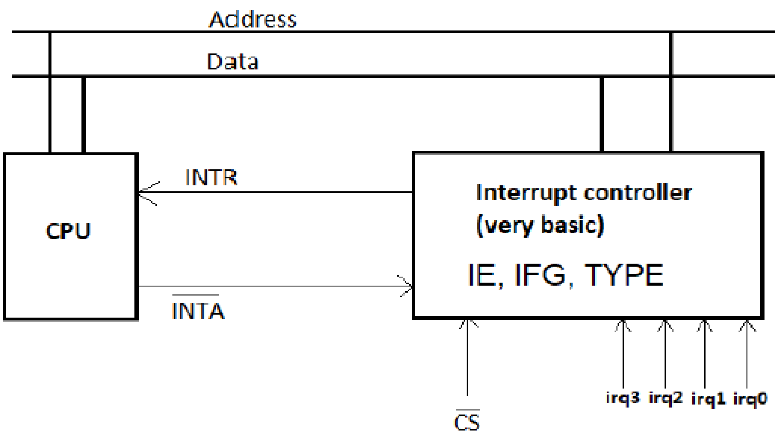


3. interrupt controller:

This is an asynchronic peripheral. That's means that it doesn't have a clock input, when it gets an interrupt, its sends an INTR (interrupt request) to the cpu. when it gets an INTA (interrupt acknowledge) back from the cpu, the interrupt controller will put the correct interrupt address on the address bus, and will save the next Pc in $ra register (like in a jal command).

After the interrupt code line, the user will need to clean the flags and return to the address in $ra using jr command.

Our CS (chip select) is the component address.



IE, Interrupt Enable Register

7	6	5	4	3	2	1	0
				BTIE	KEY3E	KEY2E	KEY1E
				rw	rw	rw	rw

IFG, Interrupt Flag Register

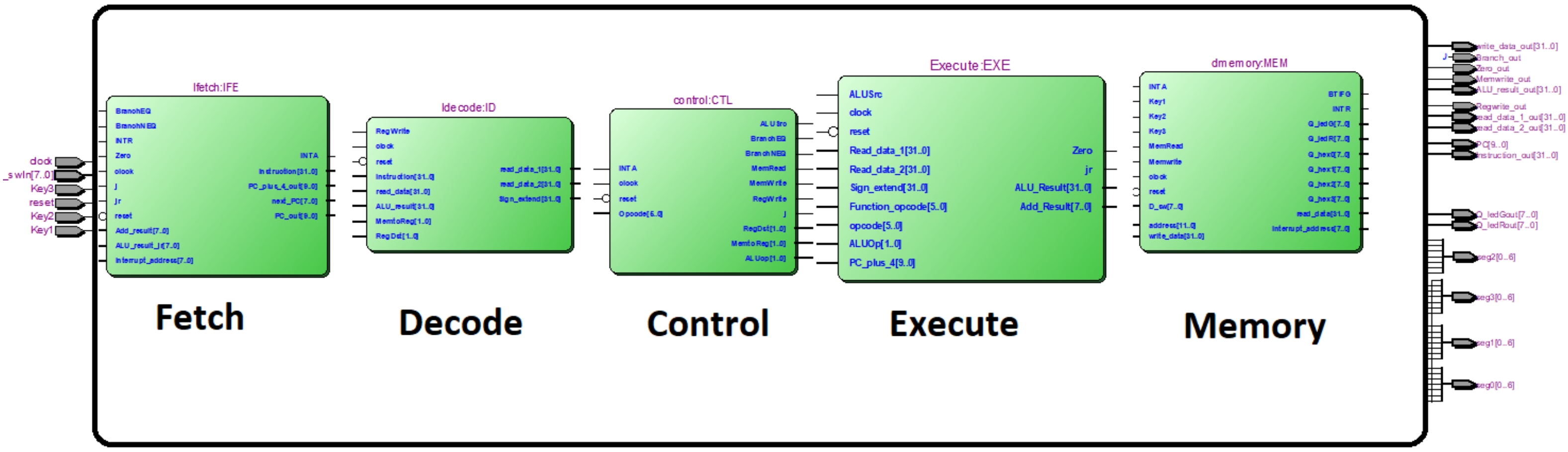
7	6	5	4	3	2	1	0
				BTIFG	KEY3IFG	KEY2IFG	KEY1IFG
				rw	rw	rw	rw

TYPE, Interrupt Type Register

7	6	5	4	3	2	1	0
0	0	0	0	TYPEx			0
				rw	rw	rw	rw

TYPE Contents	Interrupt Source	Interrupt Flag	Interrupt Priority
00h	KEY1	KEY1IFG	Lowest
04h	KEY2	KEY2IFG	
08h	KEY3	KEY3IFG	
0Ch	Basic Timer	BTIFG	Highest

Blocks diagram:



Top entity - Mips:

Our single cycle Mips is divided to 5 stages:

Fetch – getting the next command

Decode- breaking the command to an instruction

Control – manage all the control lines

Execute- calculates data using the ALU and the control lines

Memory- reads and writes data to and from memory or I/O

Mips logic usage:

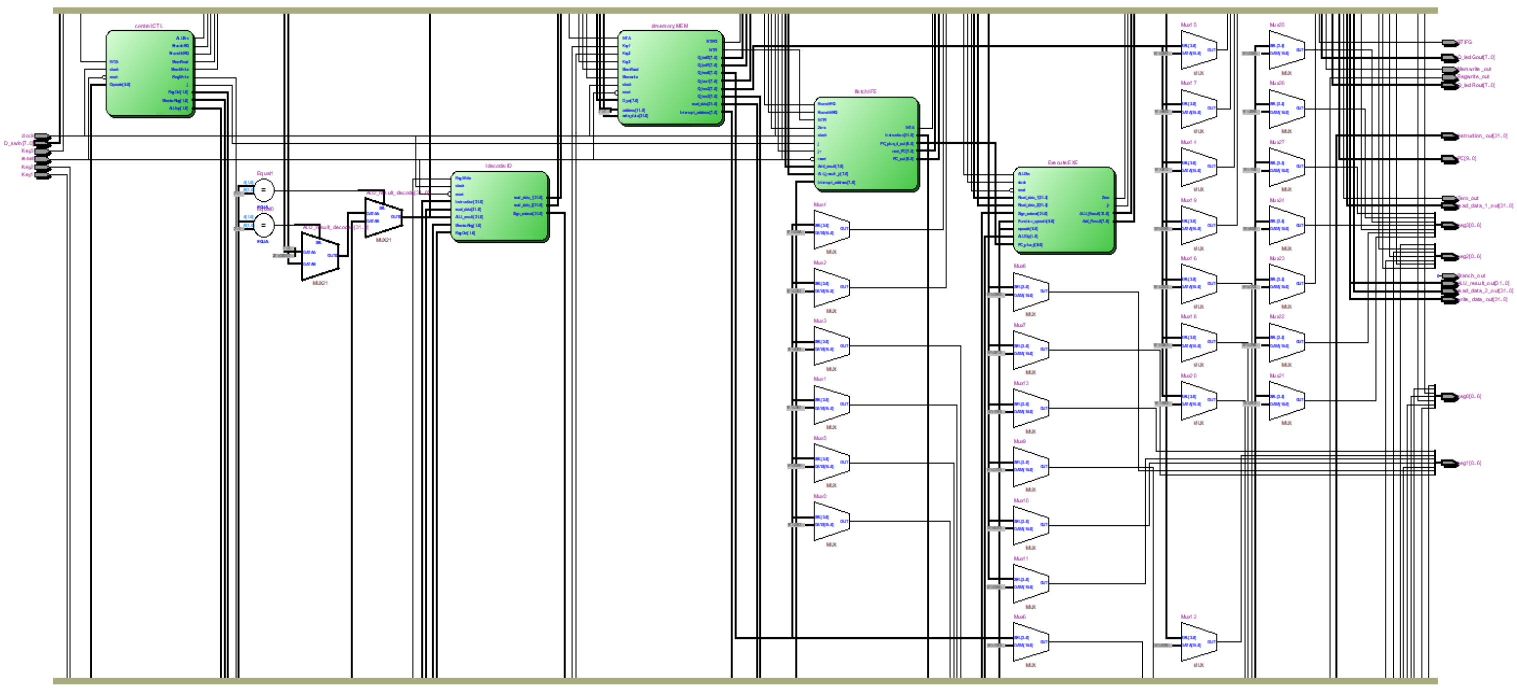
Flow Summary	
Flow Status	Successful - Fri Sep 11 15:13:22 2020
Quartus II 64-Bit Version	12.1 Build 177 11/07/2012 SJ Web Edition
Revision Name	CPUfinal
Top-level Entity Name	MIPS
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	4,882 / 18,752 (26 %)
Total combinational functions	3,340 / 18,752 (18 %)
Dedicated logic registers	2,768 / 18,752 (15 %)
Total registers	2768
Total pins	232 / 315 (74 %)
Total virtual pins	0
Total memory bits	173,056 / 239,616 (72 %)
Embedded Multiplier 9-bit elements	0 / 52 (0 %)
Total PLLs	0 / 4 (0 %)

We can see that we have used 4,882 logic elements.

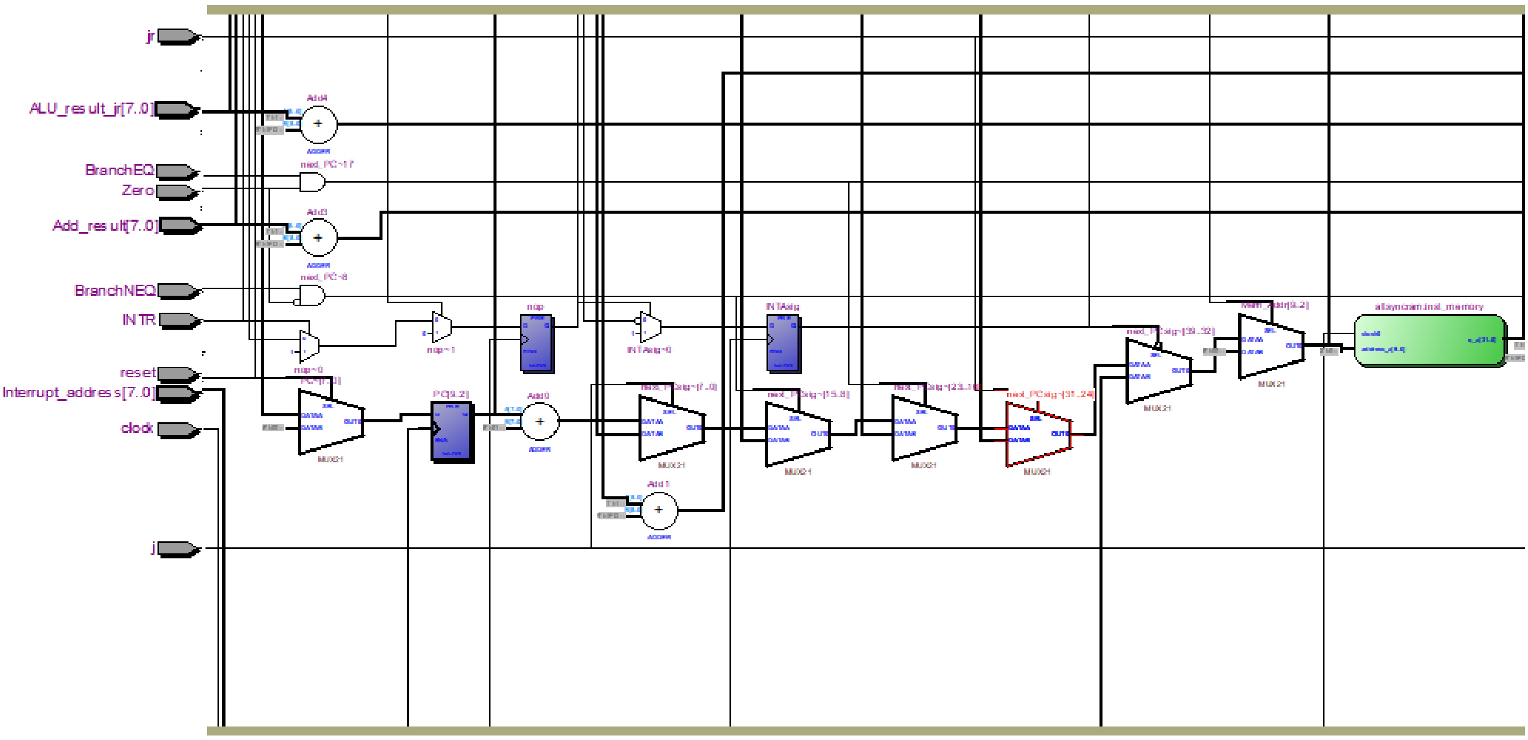
RTL View:

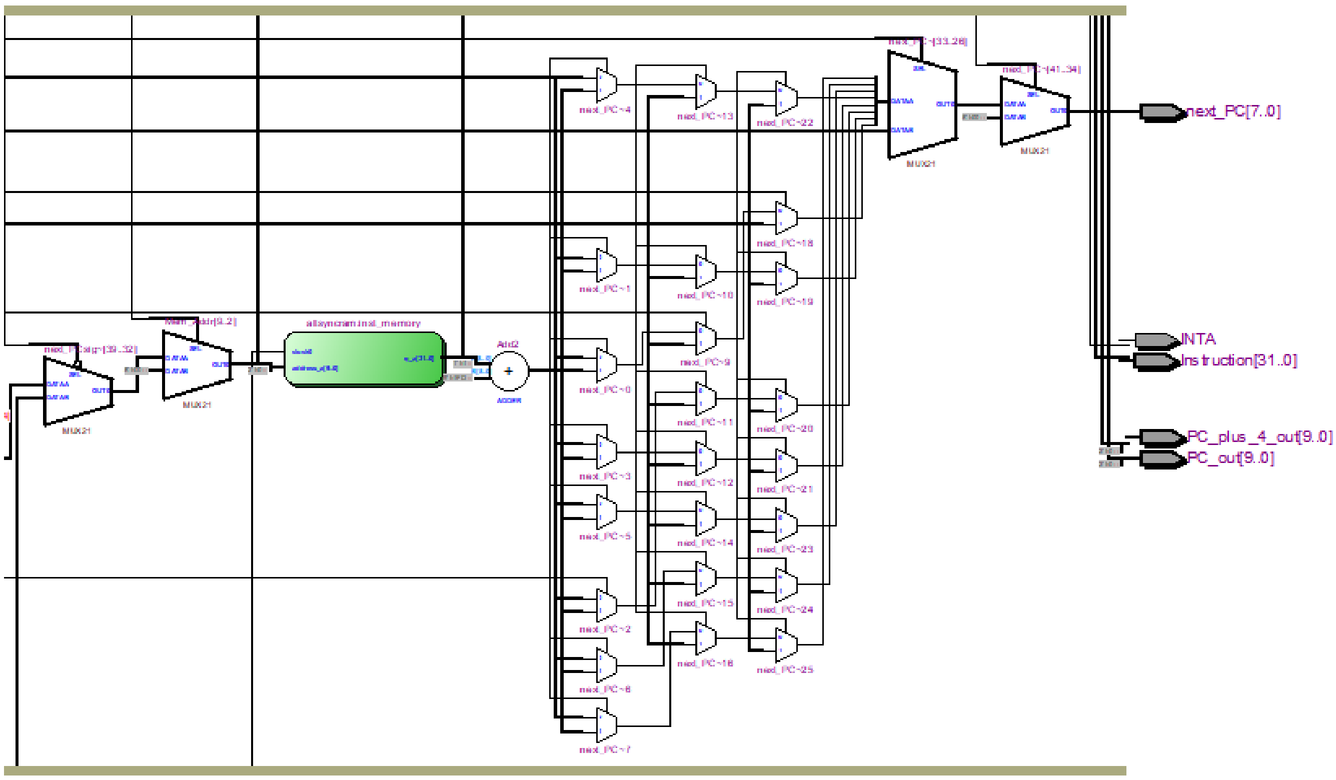
Now we **will** check each stage apart, starting from the top, Mips, and going down for each one of the stages from earlier.

Mips:

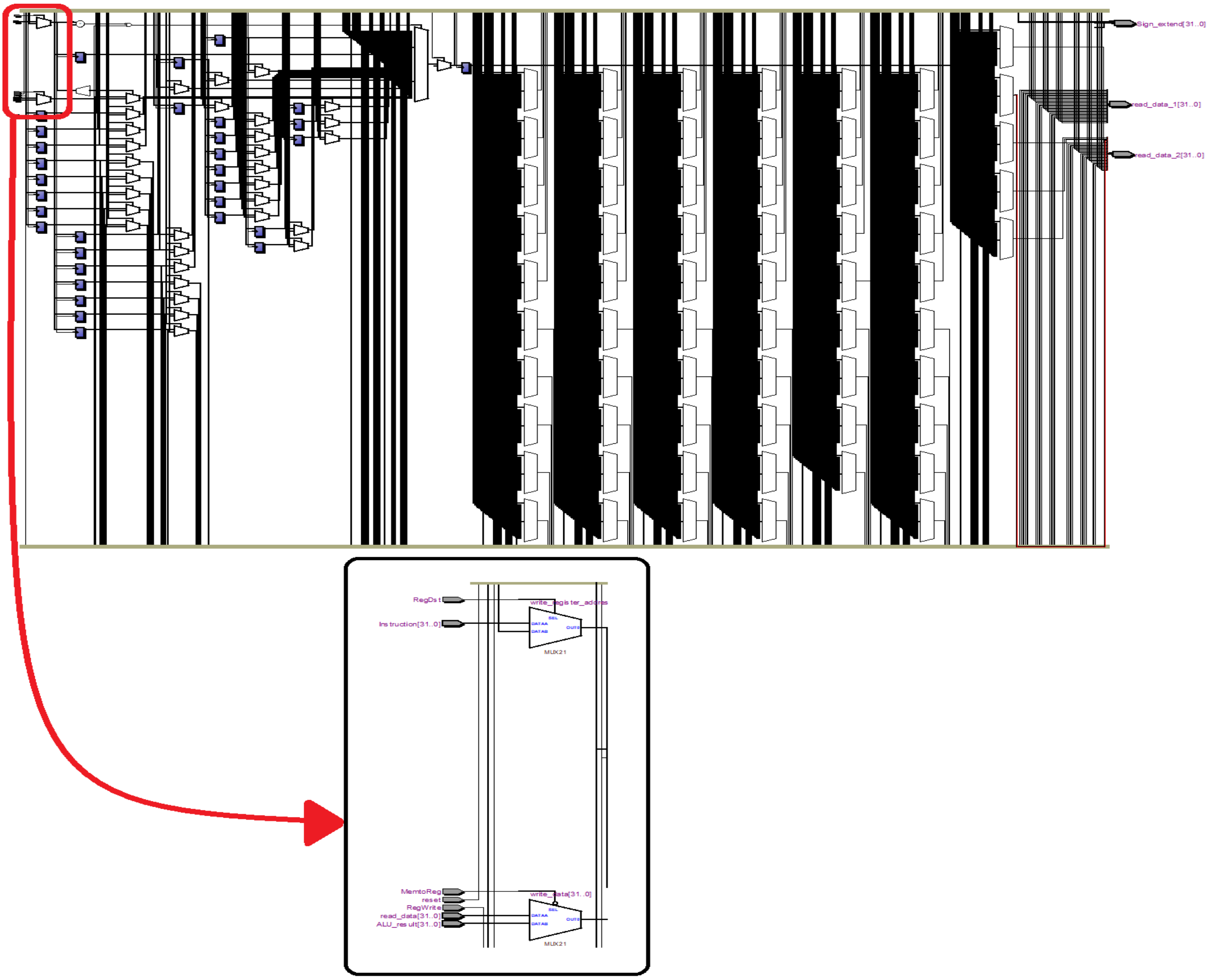


Fetch:

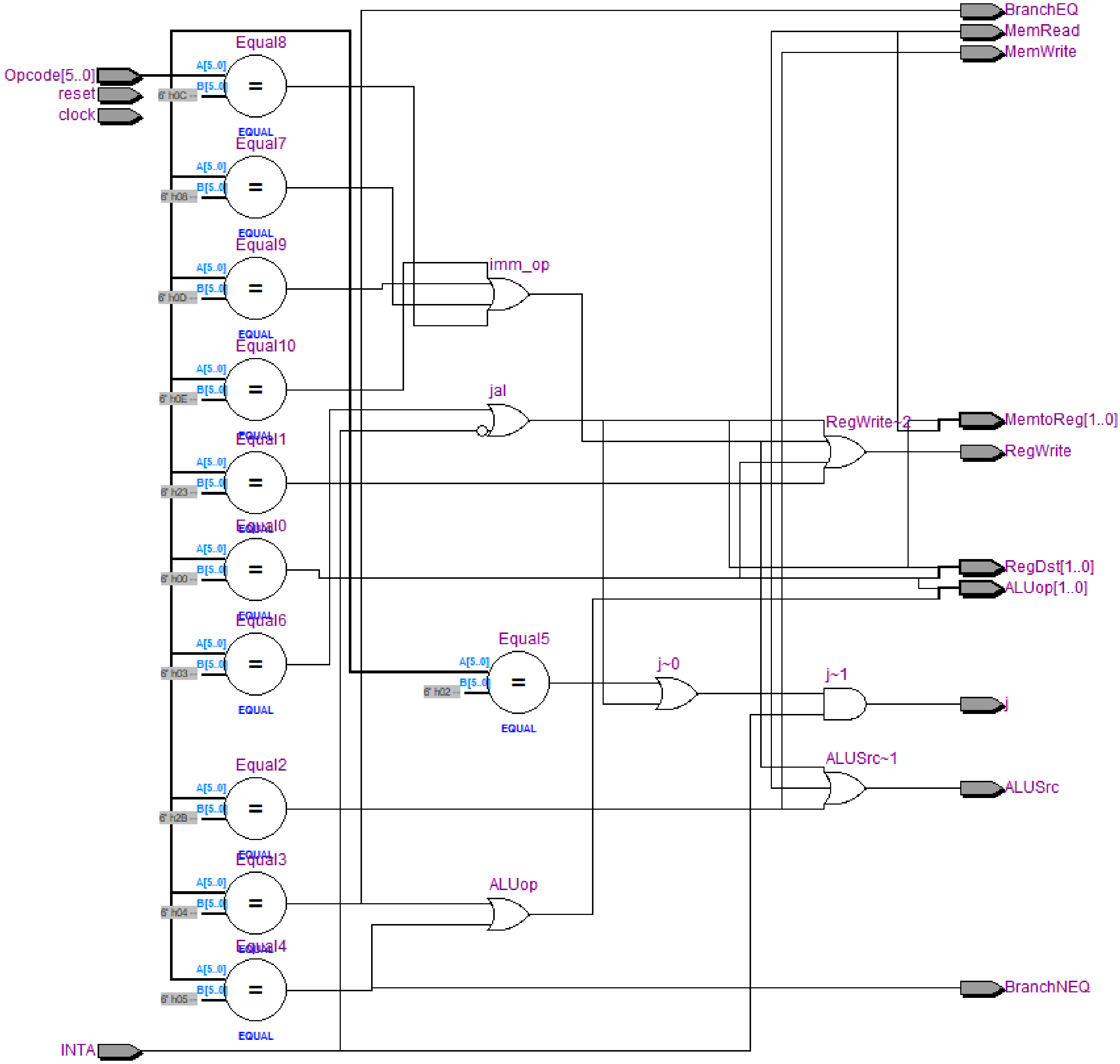

jr
ALU_result_jr[7..0]
BranchEQ
Zero
Add_result[7..0]
BranchNEQ
INTR
reset
Interrupt_address[7..0]
clock
j
altsyncram:inst_memory


next_PC[7..0]
INTA
Instruction[31..0]
PC_plus_4_out[9..0]
PC_out[9..0]
altsyncram:inst_memory

Decode:

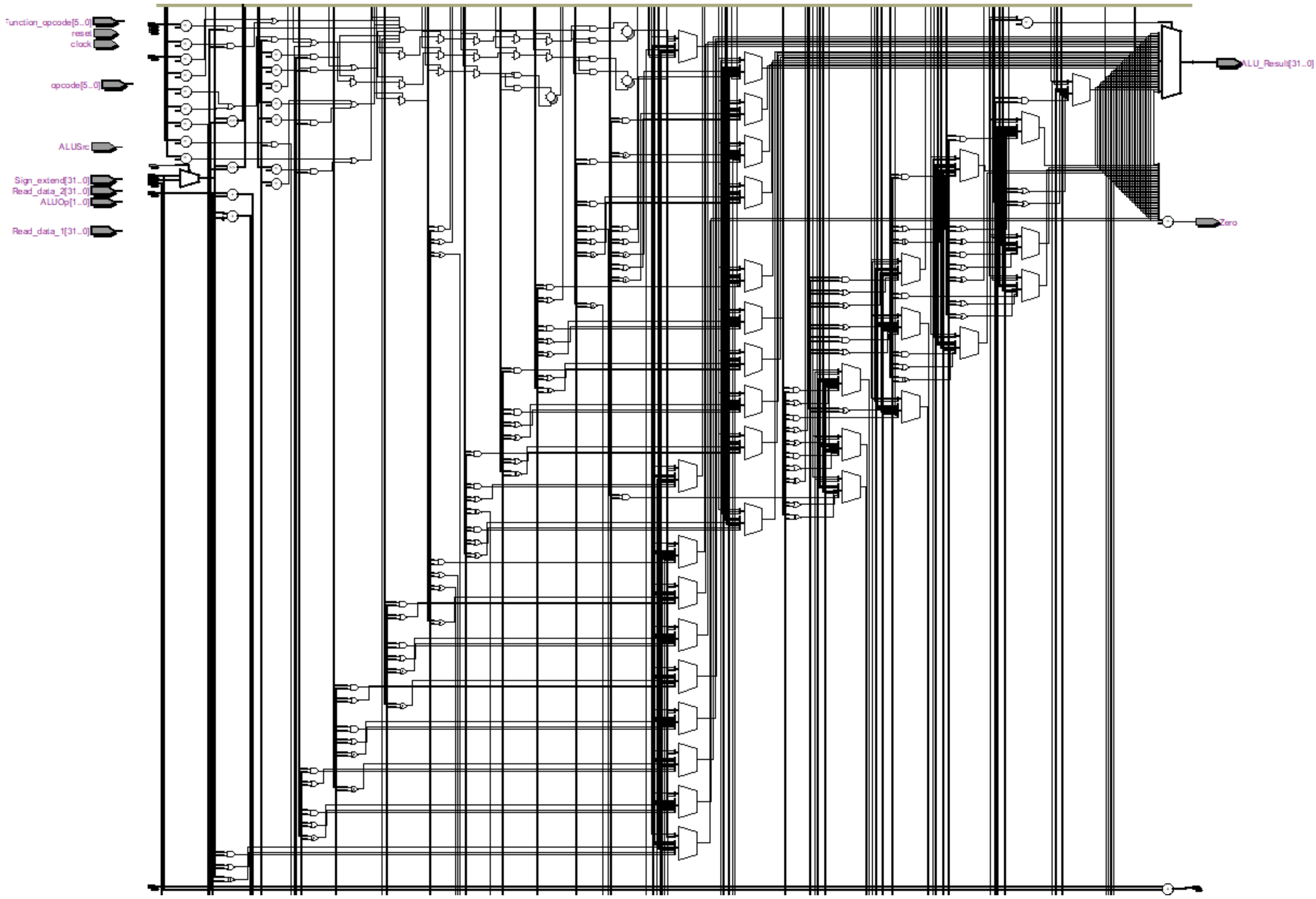

Sign_extend[31..0]
read_data_1[31..0]
read_data_2[31..0]
RegDst
Instruction[31..0]
SEL
DATAA
DATAB
OUT0
MUX21
MemtoReg
reset
RegWrite
read_data[31..0]
ALU_result[31..0]
write_data[31..0]
SEL
DATAA
DATAB
OUT0
MUX21

Control:



BranchEQ
MemRead
MemWrite
Equal8
Opcode[5..0]
reset
clock
A[5..0]
B[5..0]
6' h0C
EQUAL
Equal7
6' h08
Equal9
6' h0D
imm_op
Equal10
6' h0E
jal
Equal1
6' h23
RegWrite~2
MemtoReg[1..0]
RegWrite
Equal0
6' h00
RegDst[1..0]
ALUop[1..0]
Equal6
6' h03
Equal5
6' h02
j~0
j~1
j
ALUSrc~1
ALUSrc
Equal2
6' h2B
Equal3
6' h04
ALUop
Equal4
6' h05
BranchNEQ
INTA

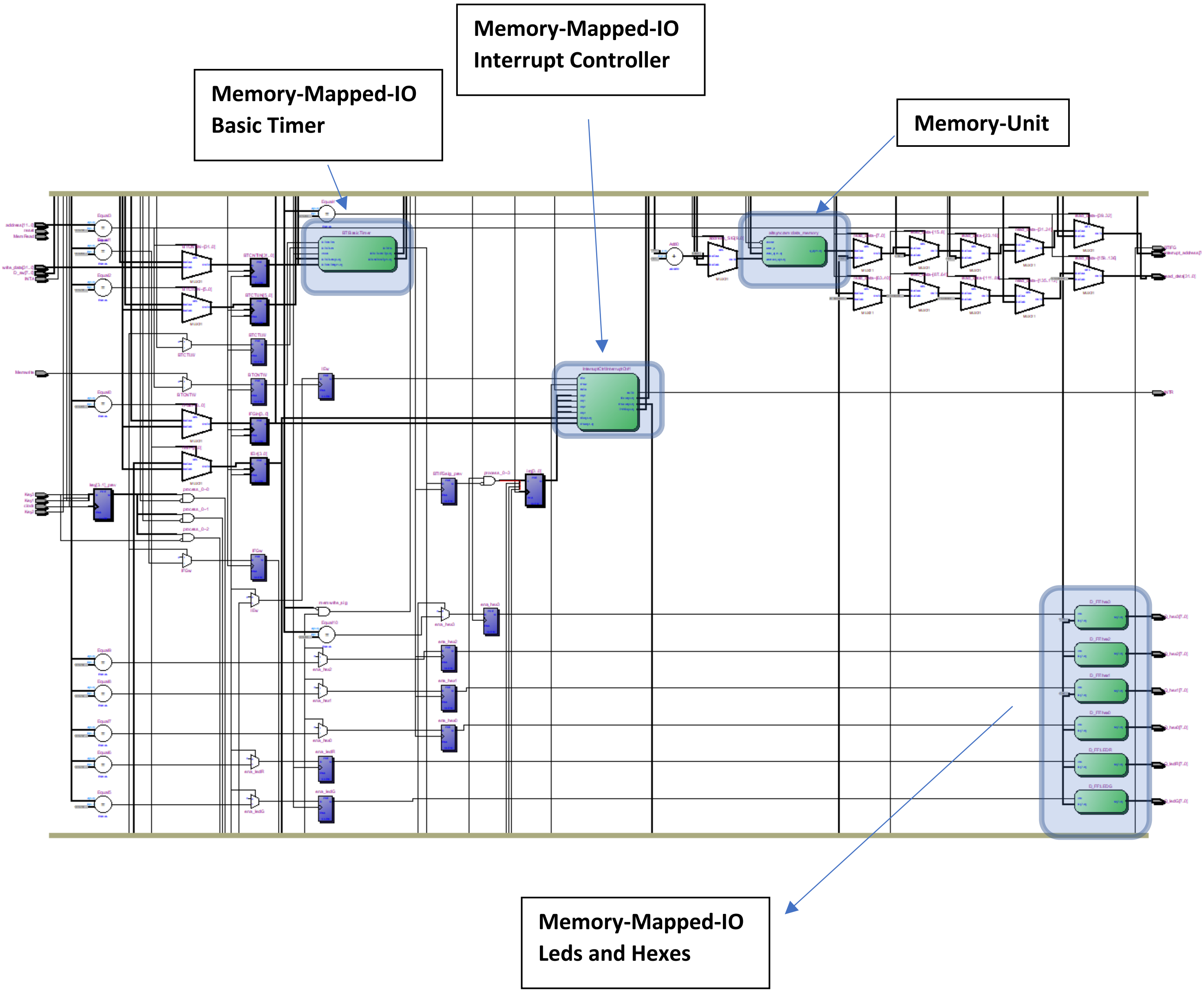
Execute:



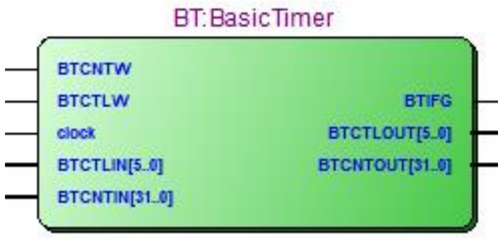
Function_opcode[5..0]
reset
clock
opcode[5..0]
ALUSrc
Sign_extend[31..0]
Read_data_2[31..0]
ALUOp[1..0]
Read_data_1[31..0]
ALU_Result[31..0]
Zero

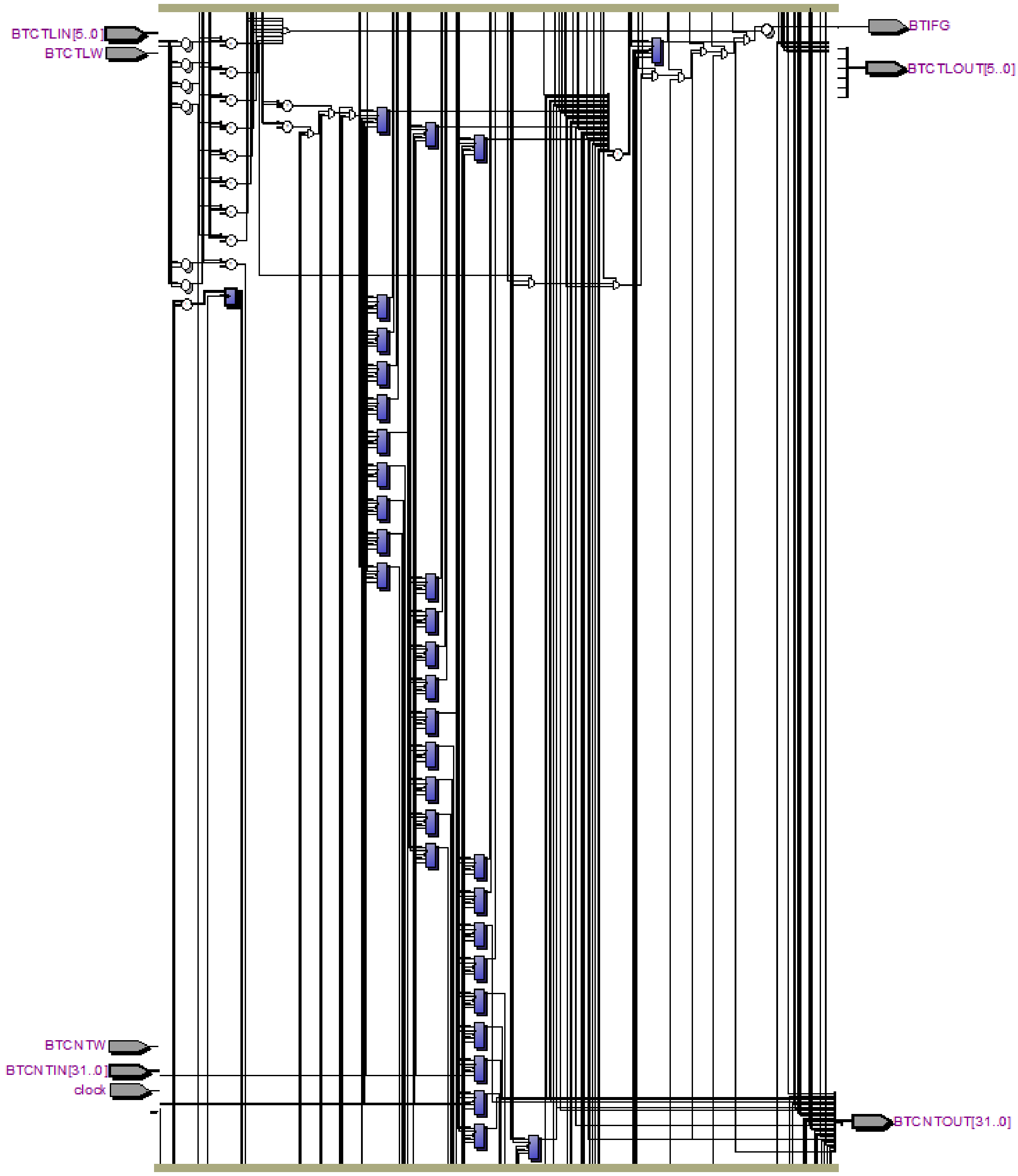
Dmemory:

Our Dmemory component is actually Memory and IO component, because we are using memory mapped io logic.

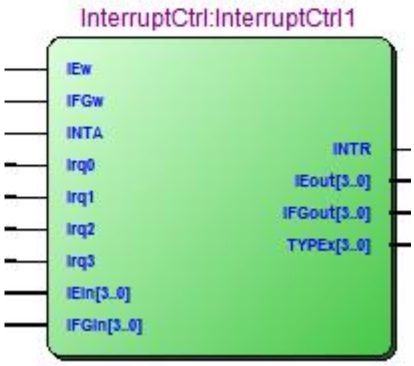


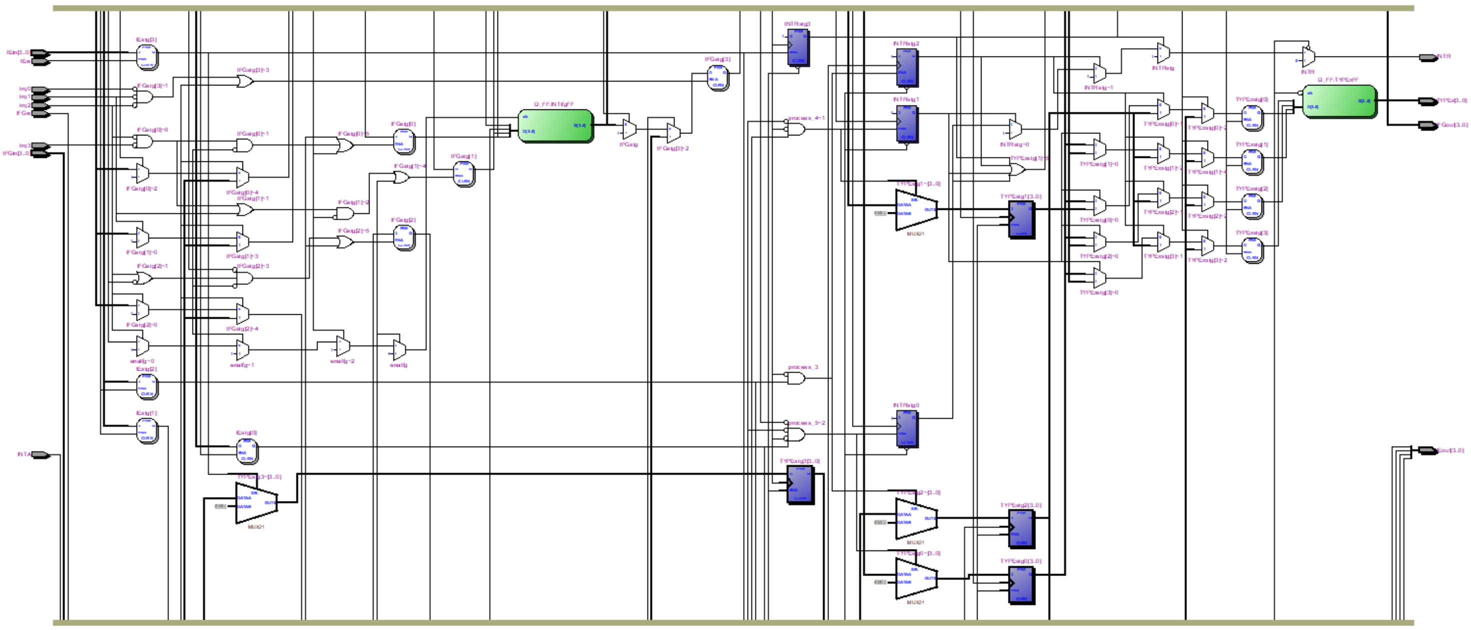
Basic Timer:


BT:BasicTimer
BTCNTW
BTCTLW
clock
BTCTLIN[5..0]
BTCNTIN[31..0]
BTIFG
BTCTLOUT[5..0]
BTCNTOUT[31..0]


BTCTLIN[5..0]
BTCTLW
BTCNTW
BTCNTIN[31..0]
clock
BTIFG
BTCTLOUT[5..0]
BTCNTOUT[31..0]

Interrup controller:





Maximal operating clock:

we have found that the maximum frequency is

Slow Model Fmax Summary

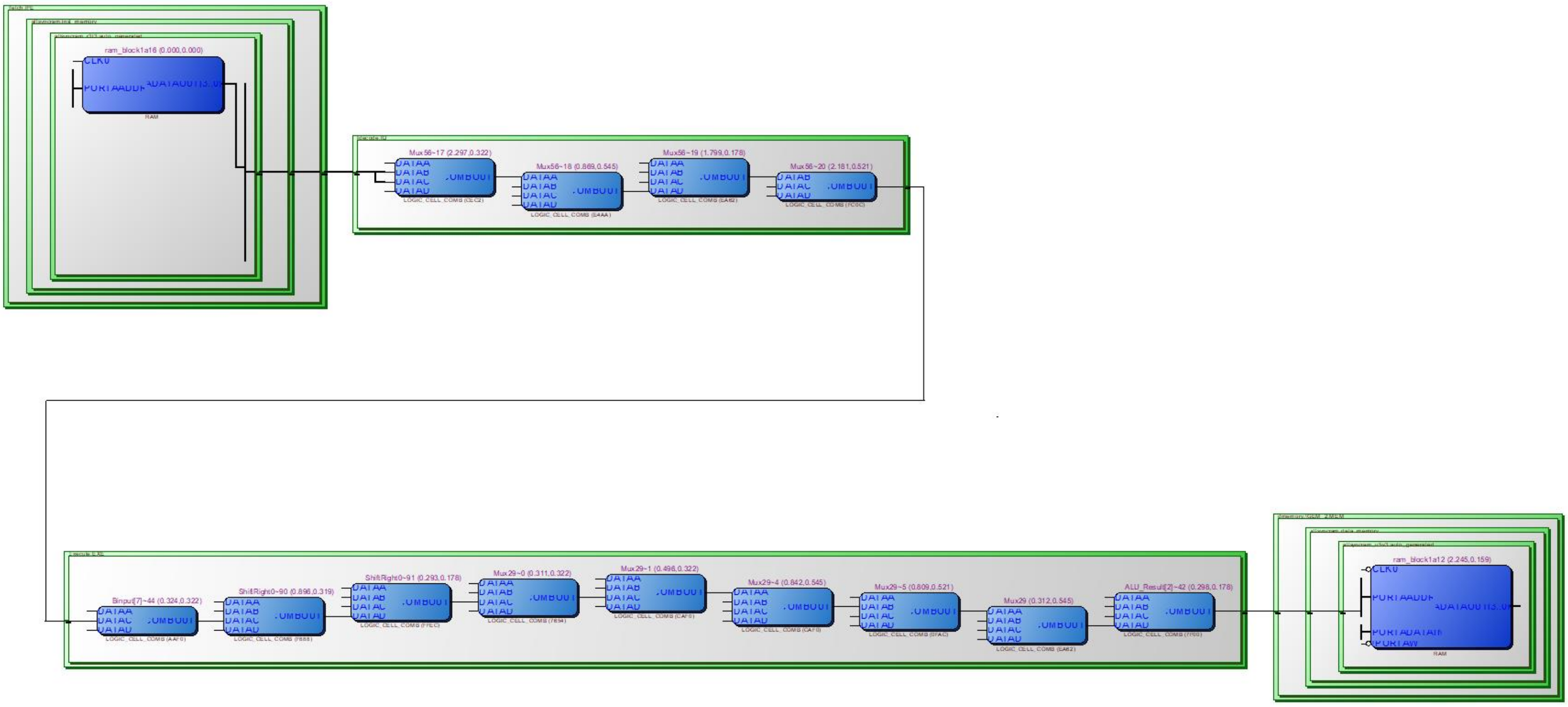
	Fmax	Restricted Fmax	Clock Name	Note
1	22.41 MHz	22.41 MHz	clock	

So the Maximal operating clock is $\frac{1}{22.41Mhz} = 44.6ns$

Longest path:

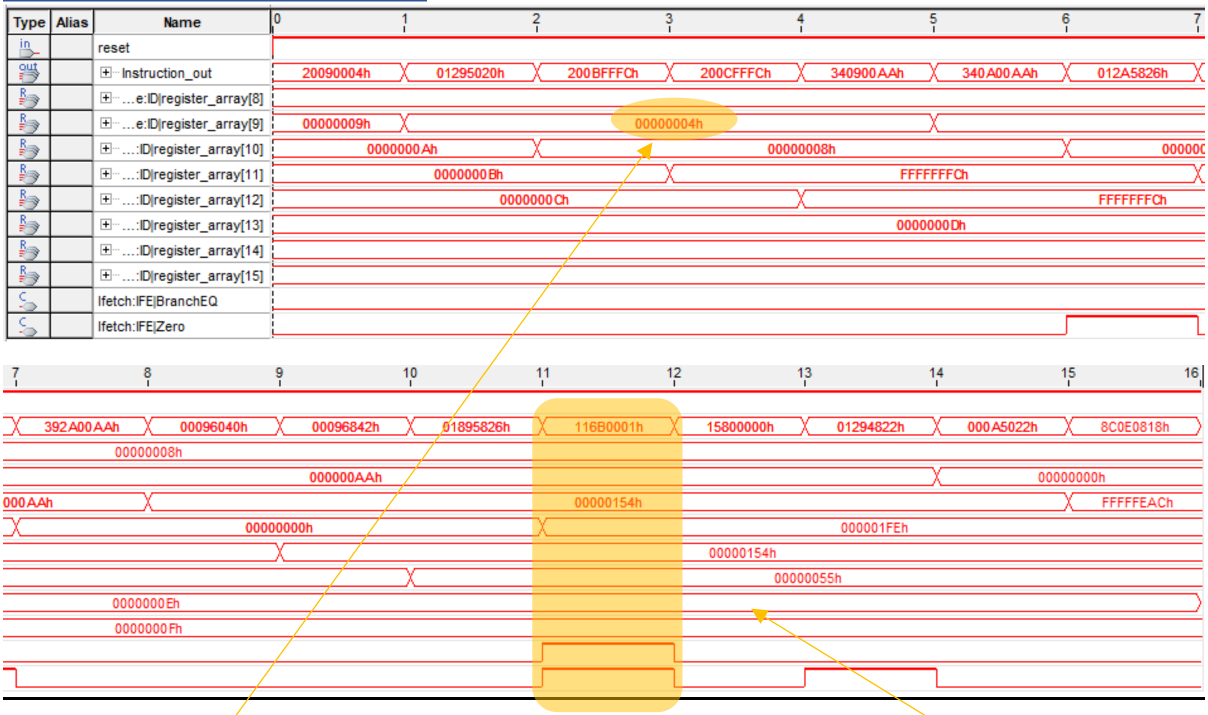
Memmory>decode>execute>memory

Will be longest on shift opcodes



Single-Tap HardWare Test:

Arithmetic Commands Test:



Beq $11,$11, check1

This command is always True, so we should jump to “check1”, as we can see. We can also see that the “BranchEq” and “Zero” control line turned on

Addi $9,0,4

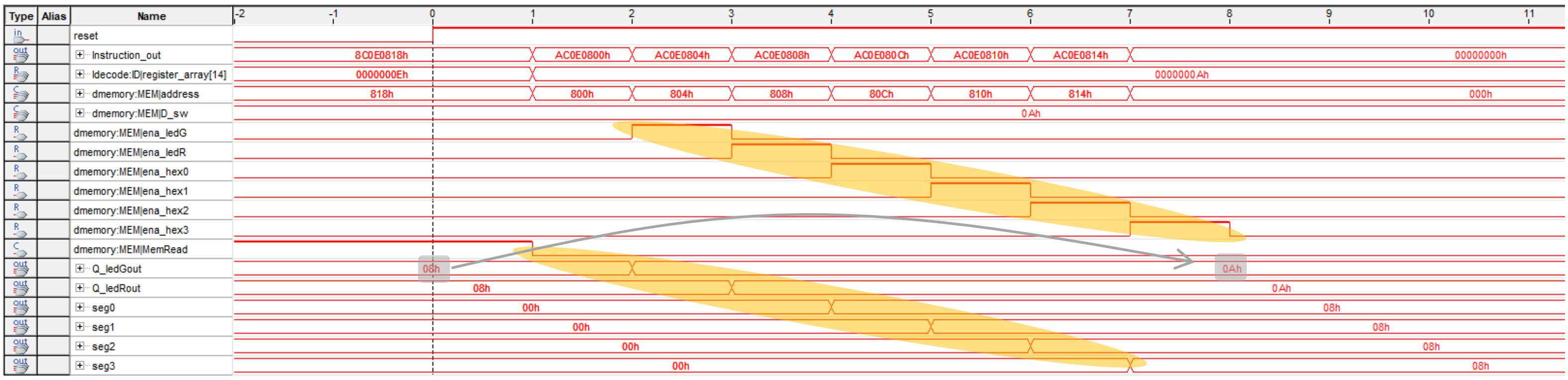
We are adding immediate 4 to register $9

The code we are running:

Address	Code	Basic	
0x00003000	0x20090004	addi $9,$0,0x00000004	5: addi $t1, $zero,4
0x00003004	0x01295020	add $10,$9,$9	6: add $t2, $t1, $t1
0x00003008	0x200bfffc	addi $11,$0,0xfffff...	8: add $t3, $zero, -4
0x0000300c	0x200cfffc	addi $12,$0,0xfffff...	9: addi $t4, $zero, -4
0x00003010	0x340900aa	ori $9,$0,0x000000aa	11: ori $t1, $zero,0xAA
0x00003014	0x340a00aa	ori $10,$0,0x000000aa	12: or $t2, $zero,0xAA
0x00003018	0x012a5826	XOR $11,$9,$10	14: XOR $t3, $t1,$t2
0x0000301c	0x392a00aa	XORi $10,$9,0x00000...	15: XORi $t2, $t1,0xAA
0x00003020	0x00096040	sll $12,$9,0x00000001	17: sll $t4,$t1, 1
0x00003024	0x00096842	srl $13,$9,0x00000001	18: srl $t5,$t1, 1
0x00003028	0x01895826	XOR $11,$12,$9	19: XOR $t3, $t4,$t1
0x0000302c	0x116b0001	beq $11,$11,0x00000...	21: beq $t3, $t3, check1
0x00003030	0x08000c0c	j 0x00003030	23: loop: j loop
0x00003034	0x15800000	bne $12,$0,0x00000000	25: check1: bne $t4, $zero, check2
0x00003038	0x01294822	sub $9,$9,$9	27: check2: sub $t1, $t1, $t1
0x0000303c	0x000a5022	sub $10,$0,$10	28: sub $t2, $zero, $t2

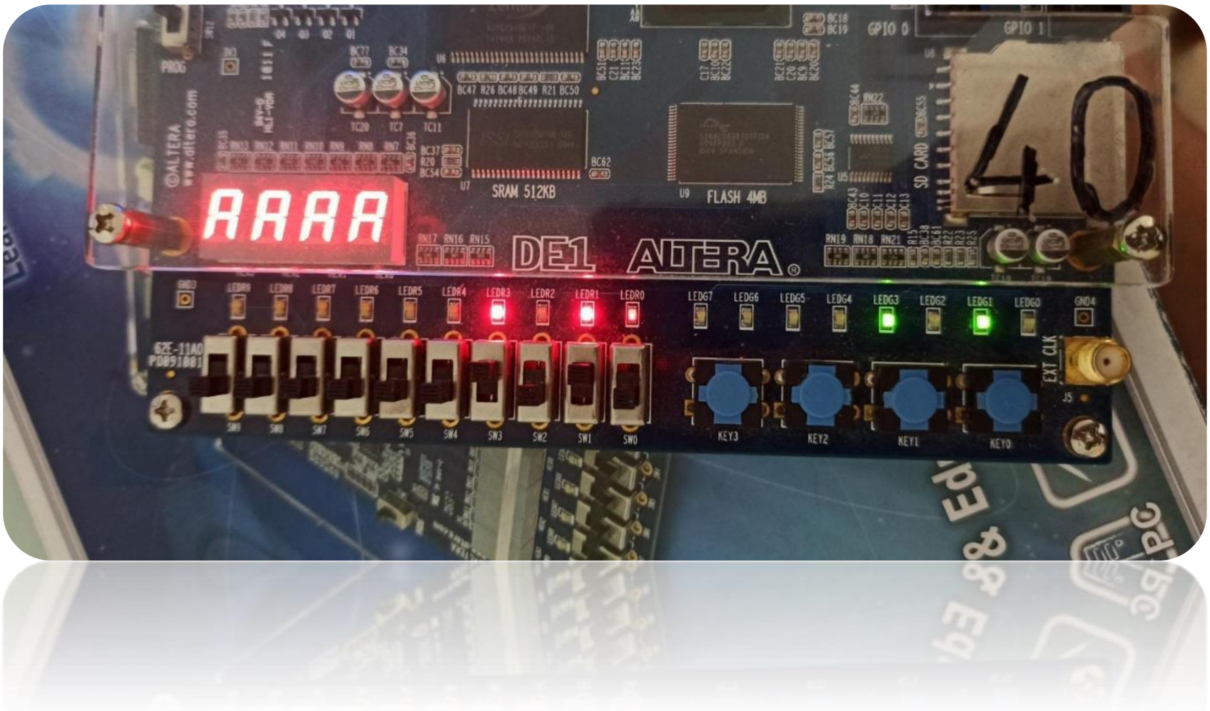
I/O Test:

we tested here the I/O, we read the data from the switches and copied it to the leds and hexes.



In this example the switches were set to 0x0A.

We can see that the enable control to each I/O is set and the value changes to 0x0A



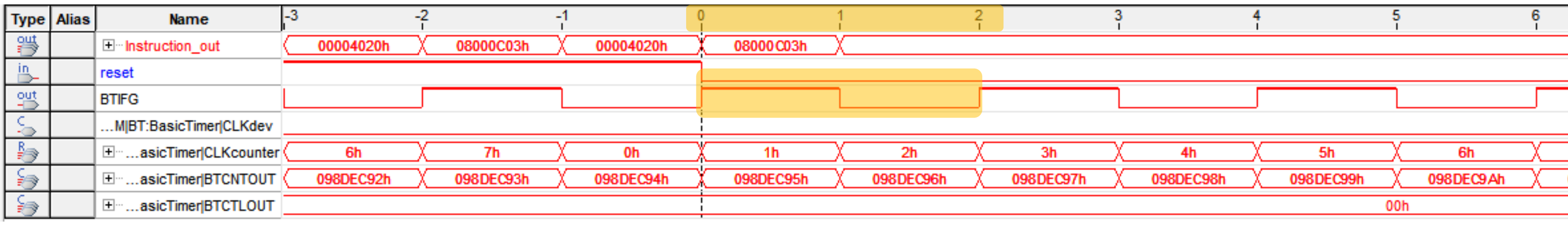
The code we are running:

Address	Code	Basic			
0x00003000	0x8c0e0818	lw $14,0x00000818($0)	6:	lw	$t6,0x818
0x00003004	0xac0e0800	sw $14,0x00000800($0)	7:	sw	$t6,0x800
0x00003008	0xac0e0804	sw $14,0x00000804($0)	8:	sw	$t6,0x804
0x0000300c	0xac0e0808	sw $14,0x00000808($0)	9:	sw	$t6,0x808
0x00003010	0xac0e080c	sw $14,0x0000080c($0)	10:	sw	$t6,0x80C
0x00003014	0xac0e0810	sw $14,0x00000810($0)	11:	sw	$t6,0x810
0x00003018	0xac0e0814	sw $14,0x00000814($0)	12:	sw	$t6,0x814

Basic timer test:

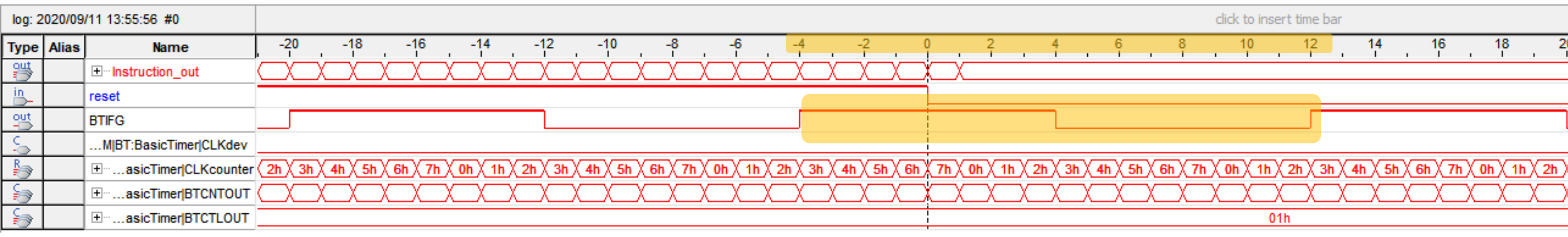
Here we tested the clock division in the basic timer.

1. BTCTL = “00000” that’s means the the clock should divided by 2



We can see that the BTIFG flag is set every 2 clock cycles.

2. BTCTL = “00001” that’s means the the clock should divided by 16



We can see that the BTIFG flag is set every 16 clock cycles.

3. BTCTL = “10001” that’s means the the clock should divided by 64

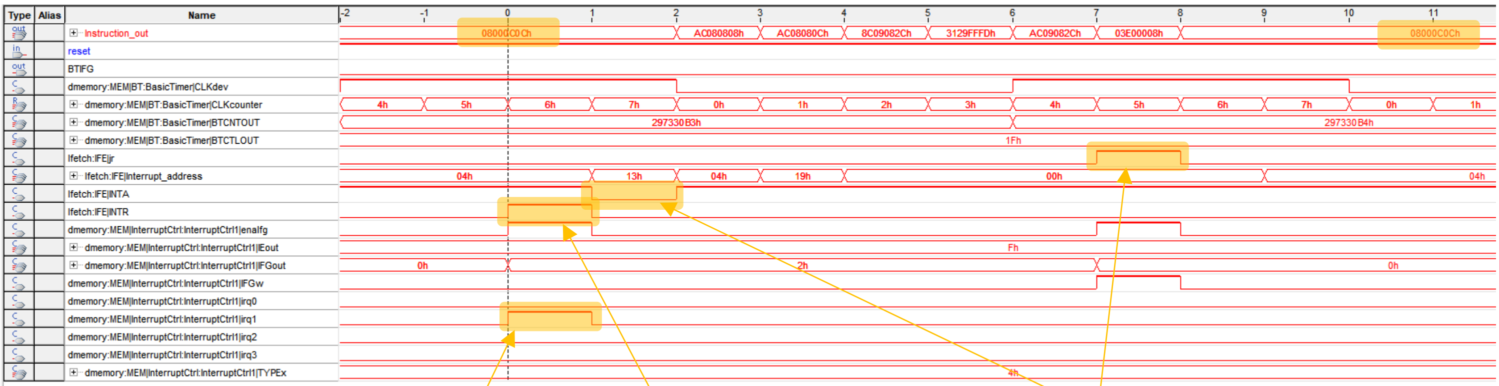
We can see that the BTIFG flag is set every 64 clock cycles. (in yellow we can see half of the clock takes 32 cycles)

Interrupt controller test:

we run this test code:

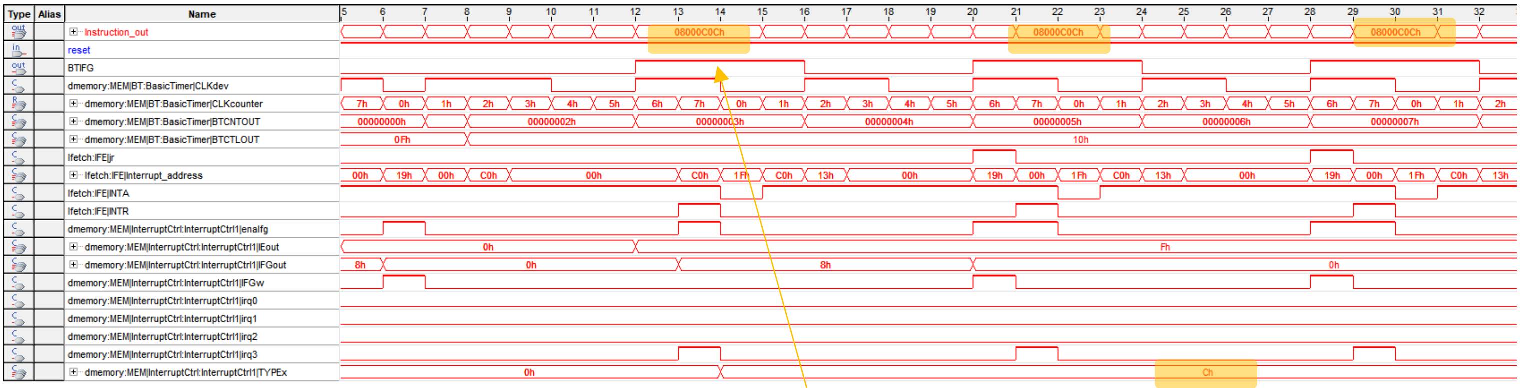
```
#-------------------- MEMORY Mapped I/O --------------------
#define PORT_LEDG[7-0] 0x800 - LSB byte (Output Mode)
#define PORT_LEDR[7-0] 0x804 - LSB byte (Output Mode)
#define PORT_HEX0[7-0] 0x808 - LSB byte (Output Mode)
#define PORT_HEX1[7-0] 0x80C - LSB byte (Output Mode)
#define PORT_HEX2[7-0] 0x810 - LSB byte (Output Mode)
#define PORT_HEX3[7-0] 0x814 - LSB byte (Output Mode)
#define PORT_SW[7-0]   0x818 - LSB byte (Input Mode)
#define PORT_KEY[3-0]  0x81C - LSB nibble (Input Mode)
#define BTCTL          0x820 - LSB byte
#define BTCNT          0x824 - Word
#------------------------------------------------------------
#define IE             0x828 - LSB byte
#define IFG            0x82C - LSB byte
#------------------------------------------------------------
.data
          N:        .word 0xB71B00
          IV:       .word KEY1_ISR
                    .word KEY2_ISR
                    .word KEY3_ISR
                    .word BT_ISR


.text
          addi $sp,$zero,0x400 # $sp=0x400
          addi $t0,$zero,0x3F
          sw   $t0,0x820       # BTIP=7, BTSSEL=3, BTHOLD=1
          sw   $0,0x824        # BTCNT=0
          sw   $0,0x828        # IE=0
          sw   $0,0x82C        # IFG=0
          addi $t0,$zero,0x1F
          sw   $t0,0x820       # BTIP=7, BTSSEL=3, BTHOLD=0
          addi $t0,$zero,0x0F
          sw   $t0,0x828        # IE=0x0F
          ori  $k0,$k0,0x01     # $k0[0] uses as GIE

          lw   $t0,0x818 # read the state of PORT_SW[7-0]
L:        j    L

KEY1_ISR: sw   $t0,0x800 # write to PORT_LEDG[7-0]
          sw   $t0,0x804 # write to PORT_LEDR[7-0]

          lw   $t1,0x82C # read IFG
          andi $t1,$t1,0xFFFE
          sw   $t1,0x82C # clr KEY2IFG
          jr   $ra # reti

KEY2_ISR: sw   $t0,0x808 # write to PORT_HEX0[7-0]
          sw   $t0,0x80C # write to PORT HEX1[7-0]

          lw   $t1,0x82C # read IFG
          andi $t1,$t1,0xFFFD
          sw   $t1,0x82C # clr KEY2IFG
          jr   $ra # reti

KEY3_ISR: sw   $t0,0x810 # write to PORT_HEX2[7-0]
          sw   $t0,0x814 # write to PORT_HEX3[7-0]

          lw   $t1,0x82C # read IFG
          andi $t1,$t1,0xFFFB
          sw   $t1,0x82C # clr KEY3IFG
          jr   $ra # reti

BT_ISR:   addi $t0,$t0,1  # $t1=$t1+1
          sw   $t0,0x800 # write to PORT_LEDG[7-0]

          lw   $t1,0x82C # read IFG
          andi $t1,$t1,0xFFF7
          sw   $t1,0x82C # clr BTIFG
        jr   $ra # reti
```

in the first test we pushed key2, after reaching to the main loop.



we can see the key pressed, and then an INTR is sent to the CPU, we receive back INTA and put the IV (interrupt Vector) address on the address bus. When we finished we call jr command and go back to the main loop.

In the second test we've tested the Basic Timer interrupt (with a very fast clock):



We can see that each time the interrupt finished it return to the loop.

We can see that the Type changed to 0xC.

In the next test we can see all the interrupts of keys 1-3 and the Basic Timer in the wave form of ModelSim: