Johnathon Karcz
NetID: karcz2

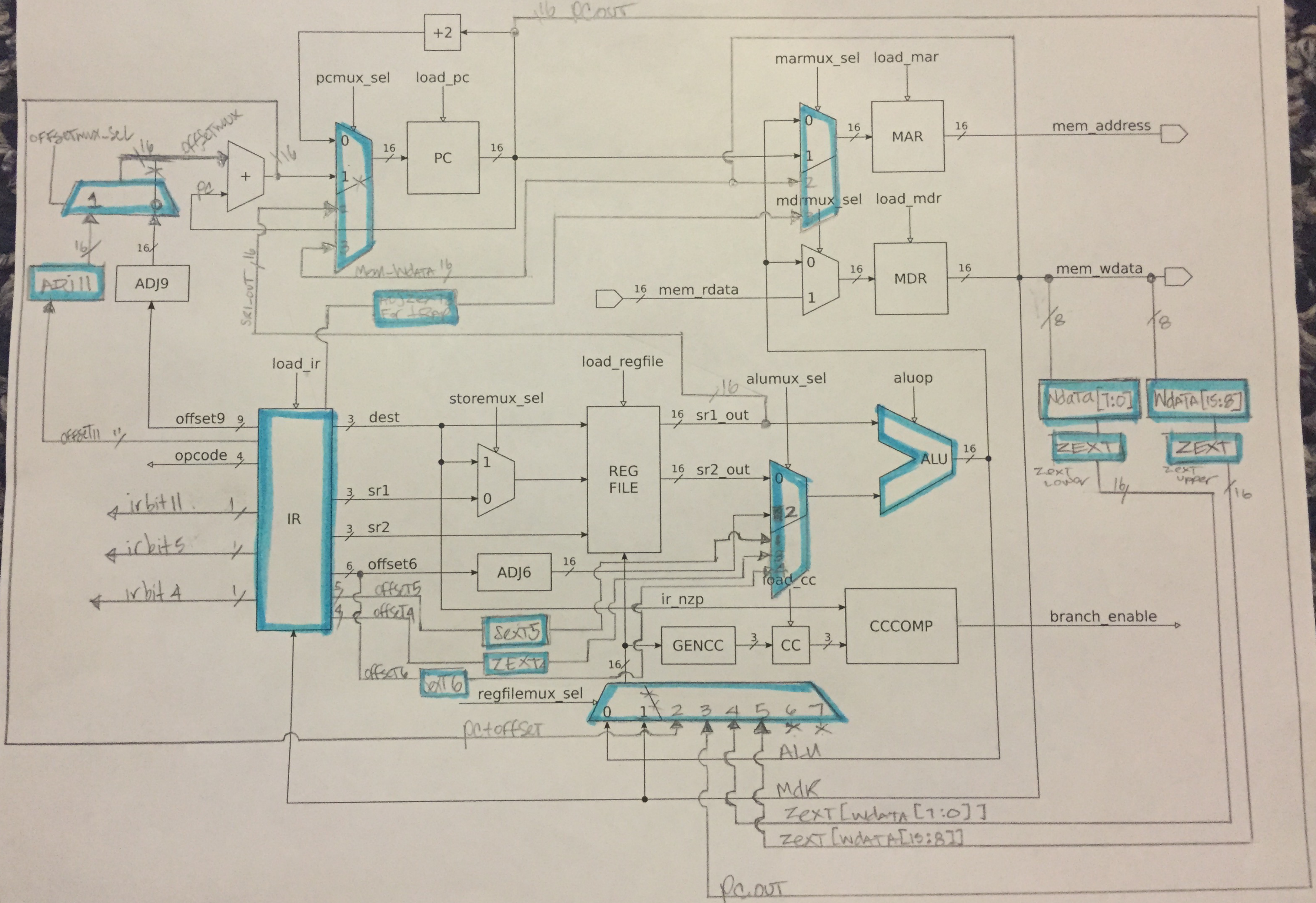
ECE411

MP1

Final Hand In

09/13/2015

Paper Design of your Datapath



+2
pcmux_sel
load_pc
PC
16
16 PC OUT
marmux_sel
load_mar
MAR
mem_address
mdrmux_sel
load_mdr
MDR
mem_wdata
16 mem_rdata
OFFSETMUX_SEL
OFFSETMUX
PC
+
ADJ11
ADJ9
SR1_OUT 16
Mem_Wdata 16
load_ir
IR
offset9 9
OFFSET11 1
opcode 4
irbit11 1
irbit5 1
irbit4 1
dest 3
sr1 3
sr2 3
offset6 6
offset5 5
offset4 4
storemux_sel
load_regfile
REG FILE
sr1_out
sr2_out
ADJ6
SEXT5
ZEXT4
offset6
SEXT6
alumux_sel
aluop
ALU
load_cc
ir_nzp
GENCC
CC
CCCOMP
branch_enable
regfilemux_sel
0 1 2 3 4 5 6 7
PC+OFFSET
ALU
MDR
ZEXT[WDATA[7:0]]
ZEXT[WDATA[15:8]]
PC_OUT
WDATA[7:0]
WDATA[15:8]
ZEXT
ZEXT LOWER
ZEXT UPPER
8

Description of Highlighted Blocks

ADJ11

The ADJ11 block is a reinstantiation of the given ADJ module, where the width parameter is set to 11. This allows for an 11 bit input from the IR to be shifted and then sign extended to 16 bits. This signal is then added to PC via an adder.

Used with:
JSR instruction: Supplies the offset to be added to PC from instruction, to the subroutine.

ADJZEXT8

The ADJZEXT8 block is similar to the ADJ11 or any of the other ADJ modules. However, this is an altered version that zero extends apposed to sign extending. This block will take the [7:0] bits from the Instruction out of IR and send it to the MAR in order to retrieve an instruction from memory.

Used with:
TRAP instruction: Supplies address of trap instruction in memory. This will be loaded into the PC.

SEXT5

The SEXT5 block is a custom module used to sign extend the signal out of the IR. The block accepts a 5 bit signal from the IR[4:0] bits and sign extents them without shifting.

Used with:
Immediate ADD: The ADD instruction has the option to add an immediate value determined by the IR[4:0] bits.
Immediate AND: The AND instruction has the option to AND the value in the register against a sign extended 5 bit value determined by the IR[4:0] bits.

SEXT6

The SEXT6 block is a reinstantiation of the SEXT module used for SEXT5. The block accepts a 6 bit signal from the IR[5:0] bits and sign extents them without shifting.

Used with:
STB: Used to determine the offset from the base register.
LDB: Used to determine the offset from the base register.

ZEXT4

The ZEXT4 block is a custom module that takes an input signal and zero extends it to 16 bits.

Used with:
SHF: This module is exclusively used for the Shift (SHF) instruction, in order to determine the number of shifts to execute. The shift instruction however, has 3 possible executions. RSHFL: Logical Right Shift, RSHFA: Arithmetic Right Shift, and LSHF: Left shift, where clearly left shifting would always be logical. We do not need to shift in the negative direction, as there is a shift for both directions. Thus, zero extending the immediate value is ideal here to generate only positive numbers.

ZEXTLOWER & ZEXTUPPER

The ZEXTLOWER and ZEXTUPPER blocks are instantiations of the earlier noted ZEXT module, both take an input of 8 bits, and output a 16 bit unsigned signal.

Used with:
LDB: LDB will take the data from the MDR and load a single zero extended byte of that data to the destination register. The MDR[15:8] and MDR[7:0] signals are both zero extended and sent to the regfilemux so that either one could be selected based on the address of the MAR used to retrieve the data.

ALU

The ALU has been altered, not created. After noticing that there was an available selection bit for the ALUOP selection. I decided to add an ALU operation that takes the data in the source register SR1[7:0], and copy it to the [15:8] bits as well before sending out to be used by another module.

Used with:
STB: So far, that main use for this added operation to the ALU is for the STB instruction. This is to keep control logic out of the memory and in the Control Block of the CPU. The memory module discards the LSB of the address. so you can not simply offset the address and write to the lower byte in order to write to the upper byte of a memory address. As a solution, the data copied into the MDR has the same data for [7:0] and [15:8]. When writing to an odd address. The STR1[7:0] data can then be written to the MEM[15:8] (odd numbered) address location.

IR

There are two main alterations made to the IR unit. First would be additional outputs of bits used for specific instruction signals. This includes any immediate values, IR[5:0], IR[4:0]

IR[10:0], ect. As well as some specific bits used for control logic, IR[11], IR[5], IR[4]. The second adjustment made to the IR would be the addition of some selection logic where the destination register is set to R7 for instructions that need to store PC into R7.

Used with:
To avoid being overly redundant, I will summarize the usage of the outlined changes by noting that the immediate values used are noted above as they all are either sent to a ZEXT, SEXT ADJZEXT or ADJ module for extending. Another way of noting this would be that all instructions that use an immediate value will use this alteration. Which is most of them.
The Desination output alteration is used for the JSR, JSRR, and TRAP instructions. This is so that we can store the PC value before jumping and then return back to the next instruction after the subroutine execution is complete.

Explain How You Tested Your Design

I decided to design and implement all of my modules and then test. Since all of the instructions were written at the time that I started testing, I decided that it would be easiest to test instructions in order of relation. For example, you will see in the test code that I provided that I started by testing all of the load and store operations in a series. The test code that I had written does not do anything useful. The instructions are written in such a way, that they provide me with easy to follow signals that I can monitor via the waveforms. For example, I might STI a value into a memory location, and then LDI to check if it was stored correctly, and then LDR to double check that it was stored and make sure that I did not incorrectly write my test code.

For the most part, I would monitor the changing values of registers to make sure that they do what I intended them to do. I also monitor the registers to make sure that no other registers are not unintentionally changing.

A mild exception to this is a block where I tested condition codes. This was a series of instructions that would force the loading of a CC. Once loaded, I would then immediately after have a BR instruction, right after that would be a LOAD instruction with a very specific (`0x0BAD`) value. The point of this series, is that if i noticed (`0x0BAD`) pop up in the wave forms, I knew there was a problem, as the branch instruction should jump over the load.

This only helps you find a problem. There is more involved once a bug is found. Typically, I like to trace backwards or forwards through the signals to see where there is an interruption in expected signals or values. For this reason it is important to write test code that covers the cases, but is easy to follow through the datapath. Typically it is easier to trace forwards, in order to see where the signal starts to go bad. This gives you a point in your code to start investigating. This is also where the drawing of the datapath and outline from the ISA become invaluable. Tracing the path and checking to see what needs to be turned on and off would be impossible without these resources.

I also used the memory list and the ability to run the program for specific durations in order to check the contents of memory. For example. If i wanted to check the result of a store operation at 1500ns. I restart -f, in model sim and then run 1500ns. the memory in the memory list will now show values at 1500ns. I can then check the memory location in question to confirm that the data was stored as desired. This is good when you do not want to rely on your load instructions to check the result of a store instruction.

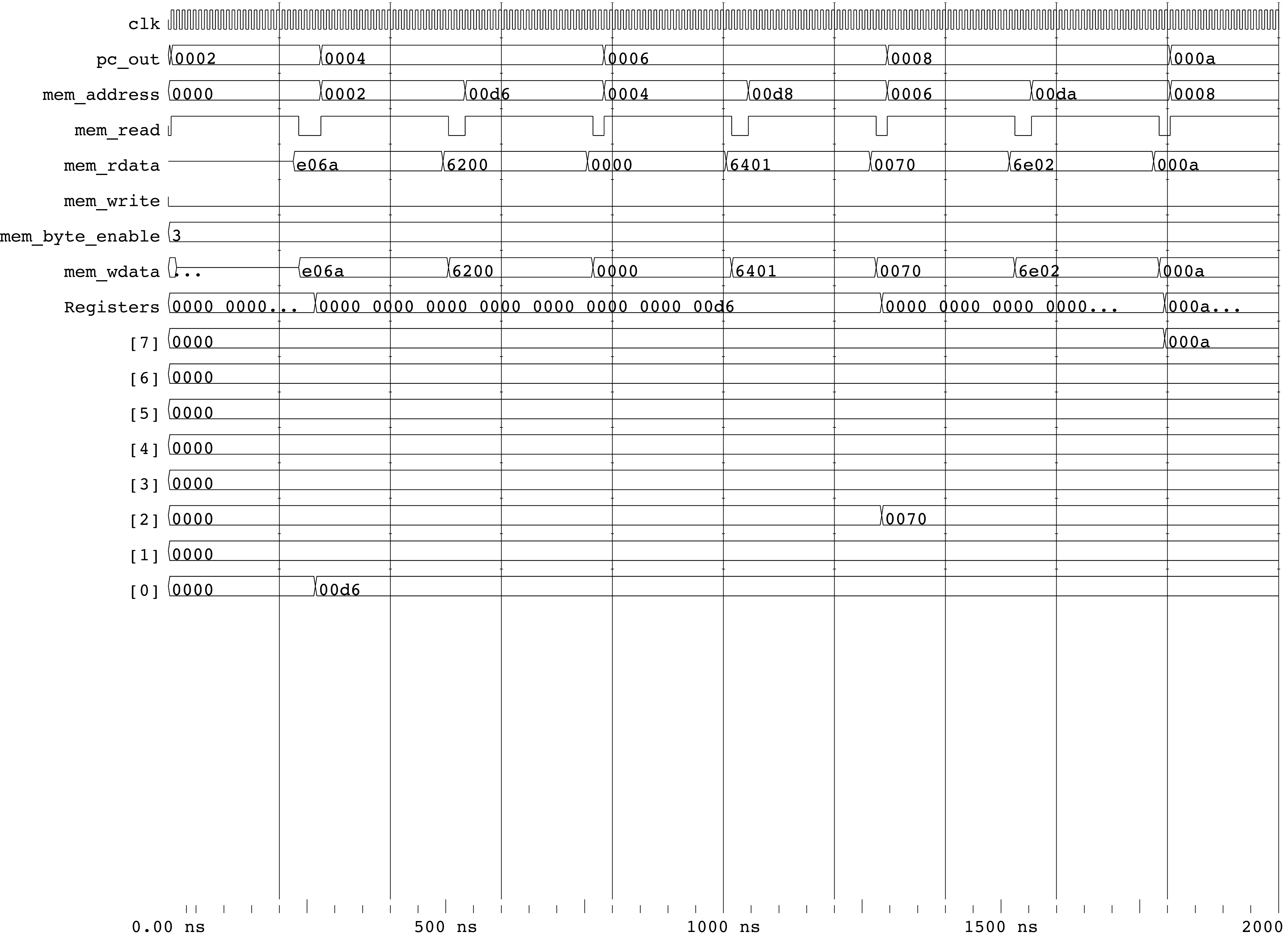
Since my personally written test code covers all of the instructions, and condition code behaviors. I feel that i have successfully covered all of the cases needed to run the provided test code successfully. I then traced through the provided code as a sanity check to make sure that i did not incorrectly write the test code itself. Each instruction in both sets executed as I expected.

To summarize, I created a test program that has a full coverage of instructions that we need to test. Then run the program and trace through making sure that the register values are behaving as programmed. If there is an issue, we use the signals at various points in the datapath to check where the problem might be. Then use that to pinpoint where the bug is written in the code.

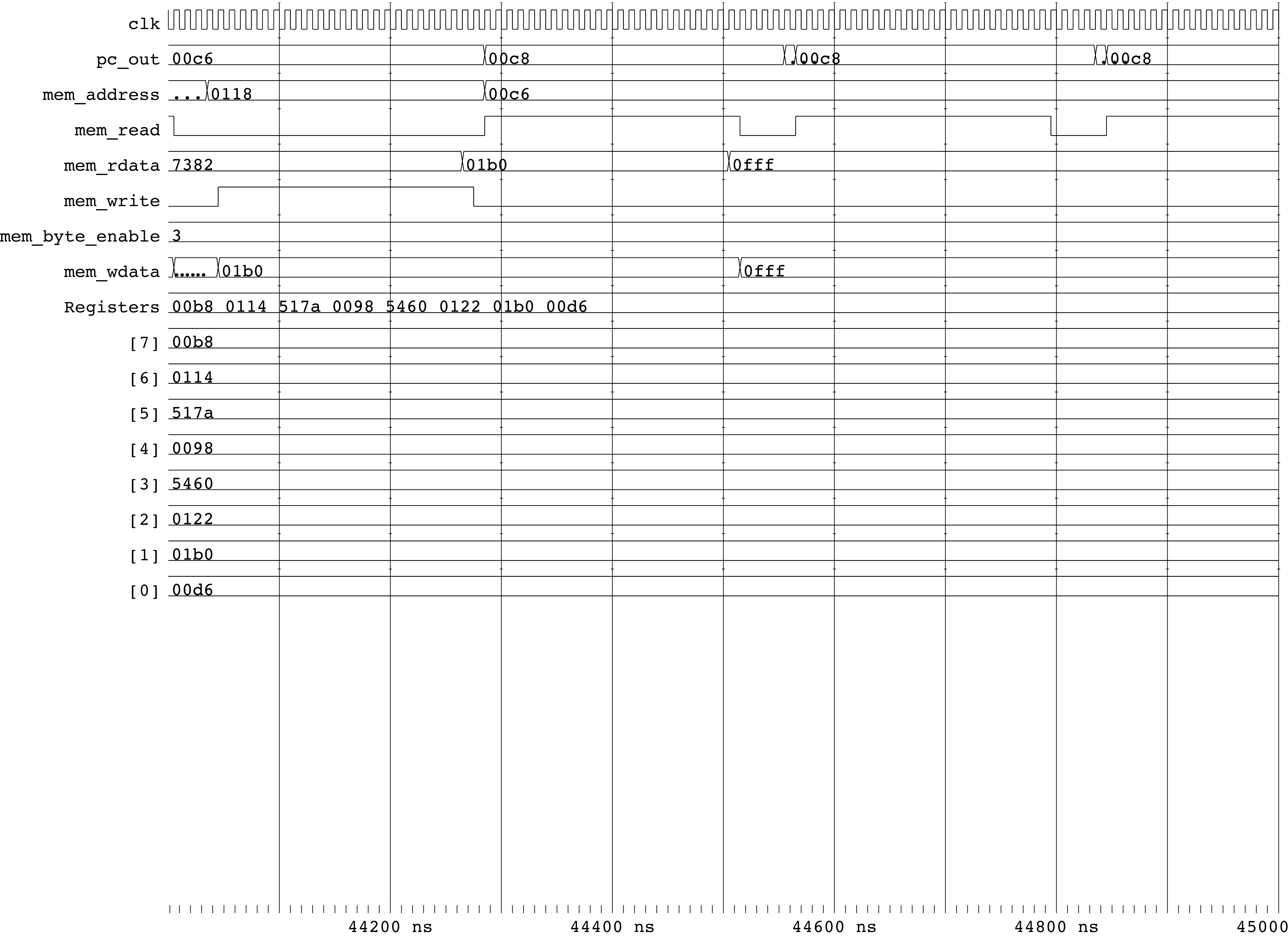
Personal Test Code

```
ORIGIN 0
SEGMENT
CodeSegment:
; initialize registers
     AND R0, R0, 0
     AND R1, R0, 0
     AND R2, R0, 0
     AND R3, R0, 0
     AND R4, R0, 0
     AND R5, R0, 0
     AND R6, R0, 0
     AND R7, R0, 0
; Test jump , we do not want that load
     LEA R1, loadstoretests
     JMP R1
     LDR R1, R0, BAD; not loading this is good,

; this section will test loads and stores.
LoadStoreTests:
     LEA R0, DataSegment  ; For addressing.
     ADD R1, R0, 1        ; for byte offset addressing.
     LEA R2, TARGET3      ; get address of target 3.
     STR R2, R0, TARGET1 ; target1 now has address of target3.
     LDR R3, R0, TARGET1 ; check to make sure R2 and R3 match,
     LDR R6, R0, COOL     ; Load R6, with C001
     STI R6, R0, TARGET1 ; Store C001 into TARGET3 Indirectly.
     AND R6, R6, 0        ; Clear R6
     LDR R6, R0, TARGET3 ; make sure that STI worked, R6 should say
'cool'
     LDI R2, R0, TARGET1 ; test LDI, R2 should also say 'cool'.
     LDB R4, R0, ABCD     ; R4 should have 00CD
     LDB R5, R1, ABCD     ; R5 should have 00AB
     ;; add data to [15:8] to make sure there are no clobering
issues.
     ; next 4 instructions.
     LDR R6, R0, EF
     ADD R4, R4, R6       ; R4 = EFCD
     LDR R6, R0, CD
     ADD R5, R5, R6       ; R5 = CDAB
     STB R5, R0, TARGET4 ; Target4 = x00AB
     LDR R6, R0, TARGET4 ; Check if that worked
     AND R6, R6, 0        ; clear it just to be thorough
     STB R4, R1, TARGET4 ; Target4 = xCDAB
     LDR R6, R0, TARGET4 ; Check if that worked

 ShiftTests:
     LDR R6, R0, ABCD     ; Start with ABCD in R6
     RSHFA R6, R6, 4      ; Should now be FABC
     RSHFL R6, R6, 4      ; SHould now be 0FAB
     LSHF R6, R6, 3       ; Should now be 7D58
     RSHFA R6, R6, 4      ; Should now be 07D5
     RSHFA R6, R6, 12     ; Should now be 0000
;; test trap by going to Condition code tests.
     TRAP CCTrap
     LDR R6, R0, BAC      ; You're BACk!
     ADD R6, R6, 4        ; Alter contents of register for tracking

     JSR jsrtest
     LDR R6, R0, BAC      ; You're BACk!
     LEA R6, jsrrtest     ; Load the addres of jsrrtest
     JSRR R6              ; jump to jsrr test.
     LDR R6, R0, BAC      ; You're BACk!

 halt:
     BRnzp halt                ; Spin

jsrtest:
     AND R6, R6, 0        ; Clear R6
     ADD R6, R6, 7        ; Add a 7 just to keep track easier
     RET                  ;

jsrrtest:
     AND R6, R6, 0        ; Clear R6
     ADD R6, R6, 8         ; Add an 8 just to keep track easier
     RET


SEGMENT    DataSegment:
ZERO:      DATA2 0
BAC:       DATA2 4xBAC
ENT:       DATA2 10
NINER:          DATA2 9999
BAD:            DATA2 4x0BAD
GOOD:           DATA2 4x600D
ABCD:      DATA2 4xABCD
TBCD:      DATA2 4x7BCD
EF:        DATA2 4xEF00
CD:        DATA2 4xCD00
TARGET1:   DATA2 0
TARGET2:   DATA2 0
TARGET3:   DATA2 0
TARGET4:   DATA2 0
COOL:      DATA2 4xC001
CCTrap:    DATA2 CCtests
; condition code tests, we will make sure that a
; CC works for each instruction that uses them.
;ADD, AND, LDB, LDI, LDR, LEA, NOT, SHF
CCtests:
     ADD R6, R6, 0         ; start with ADD to check zero condition
     BRz OK1                    ; Branch if Ok
     LDR R7, R0, BAD       ; This is the check.  we dont want to mess
up R7!
 OK1:
     ADD R6, R6, R7
     BRp OK2                    ; Branch if Ok
     LDR R7, R0, BAD       ; This is the check.  we dont want to mess
up R7!
 OK2:
     AND R6, R6, R6        ;
     BRp OK3                    ; Branch if Ok
     LDR R7, R0, BAD       ; This is the check.  we dont want to mess
up R7!
OK3:
     AND R6, R6, 0         ;
     BRz OK4                    ; Branch if Ok
     LDR R7, R0, BAD      ; This is the check.  we dont want to mess
up R7!
OK4:
     LDB R6, R0, ABCD
     BRp OK5                   ; Branch if Ok
     LDR R7, R0, BAD      ; This is the check.  we dont want to mess
up R7!
OK5:
     LDI R6, R0, TARGET1
     BRn OK6                   ; Branch if Ok
     LDR R7, R0, BAD      ; This is the check.  we dont want to mess
up R7!
OK6:
     LDR R6, R0, TBCD
     BRp OK7                   ; Branch if Ok
     LDR R7, R0, BAD      ; This is the check.  we dont want to mess
up R7!
OK7:
     LEA R6, COOL
     BRp OK8                   ; Branch if Ok
     LDR R7, R0, BAD      ; This is the check.  we dont want to mess
up R7!
OK8:
     NOT R6, R6
     BRn OK9                   ; Branch if Ok
     LDR R7, R0, BAD      ; This is the check.  we dont want to mess
up R7!
OK9:
     RSHFL R6, R6, 4
     BRp OK10             ; Branch if Ok
     LDR R7, R0, BAD      ; This is the check.  we dont want to mess
up R7!
OK10:
     RET                  ; Return to next instruction after trap
```
;;;;;;;; end of test code.

Wave Trace of the First 2000ns


clk
pc_out
0002
0004
0006
0008
000a
mem_address
0000
0002
00d6
0004
00d8
0006
00da
0008
mem_read
mem_rdata
e06a
6200
0000
6401
0070
6e02
000a
mem_write
mem_byte_enable
3
mem_wdata
...
e06a
6200
0000
6401
0070
6e02
000a
Registers
0000 0000...
0000 0000 0000 0000 0000 0000 0000 00d6
0000 0000 0000 0000...
000a...
[7]
0000
000a
[6]
0000
[5]
0000
[4]
0000
[3]
0000
[2]
0000
0070
[1]
0000
[0]
0000
00d6
0.00 ns
500 ns
1000 ns
1500 ns
2000

Wave Trace of the Last 1000ns



Entity:mp1_tb Architecture: Date: Sat Sep 12 20:54:23 CDT 2015   Row: 1 Page: 1

list of all memory writes

```
        ps       /mp1_tb/mem_address
          delta       /mp1_tb/mem_write
                 /mp1_tb/mem_byte_enable
                            /mp1_tb/mem_wdata
          0  +0                     xxxx x x xxxx
          0  +1                     0000 0 3 0000
    3155000  +5                     00f4 1 3 01e0
    3385000  +4                     00f4 0 3 01e0
    5225000  +5                     00f6 1 3 000a
    5455000  +4                     00f6 0 3 000a
    6515000  +5                     00f8 1 3 4537
    6745000  +4                     00f8 0 3 4537
   10205000  +5                     00fa 1 3 a342
   10435000  +4                     00fa 0 3 a342
   10715000  +5                     00fc 1 3 01a1
   10945000  +4                     00fc 0 3 01a1
   11225000  +5                     00fe 1 3 fe62
   11455000  +4                     00fe 0 3 fe62
   12005000  +5                     0102 1 3 0100
   12235000  +4                     0102 0 3 0100
   13025000  +5                     0100 1 3 fe6f
   13255000  +4                     0100 0 3 fe6f
   21365000  +5                     0102 1 3 001a
   21595000  +4                     0102 0 3 001a
   23205000  +5                     0104 1 3 ff9b
   23435000  +4                     0104 0 3 ff9b
   24525000  +5                     0106 1 3 000d
```

```
24755000  +4                    0106 0 3 000d
27385000  +5                    0108 1 3 517a
27615000  +4                    0108 0 3 517a
29725000  +5                    010a 1 3 0133
29955000  +4                    010a 0 3 0133
31045000  +5                    010d 1 2 9090
31275000  +4                    010d 0 3 9090
31555000  +5                    010c 1 1 acac
31785000  +4                    010c 0 3 acac
32575000  +5                    010a 1 3 90ac
32805000  +4                    010a 0 3 90ac
33355000  +5                    0110 1 3 00f2
33585000  +4                    0110 0 3 00f2
34615000  +5                    010e 1 3 0646
34845000  +4                    010e 0 3 0646
36445000  +5                    00a2 1 3 126c
36675000  +4                    00a2 0 3 126c
37225000  +5                    0112 1 3 000c
37455000  +4                    0112 0 3 000c
38785000  +5                    0114 1 3 ae85
39015000  +4                    0114 0 3 ae85
42185000  +5                    0116 1 3 600d
42415000  +4                    0116 0 3 600d
44045000  +5                    0118 1 3 01b0
44275000  +4                    0118 0 3 01b0
```

Timing Analysis Report

```
+------------------------------------------------------------------------+
; TimeQuest Timing Analyzer Summary                                      ;
+--------------------+---------------------------------------------------+
; Quartus II Version ; Version 13.1.4 Build 182 03/12/2014 SJ Full Version ;
; Revision Name      ; mp1                                               ;
; Device Family      ; Stratix III                                       ;
; Device Name        ; EP3SE50F780C2                                     ;
; Timing Models      ; Final                                             ;
; Delay Model        ; Combined                                          ;
; Rise/Fall Delays   ; Enabled                                           ;
+--------------------+---------------------------------------------------+

+------------------------------------------------------------------------------------------------------------------------
------------------------------------------------+
; Clocks
;
+------------+------+--------+-----------+-------+-------+------------+-----------+-------------+-------+--------+----
-------+------------+----------+--------+--------+---------+
; Clock Name ; Type ; Period ; Frequency ; Rise  ; Fall  ; Duty Cycle ; Divide by ; Multiply by ;
Phase ; Offset ; Edge List ; Edge Shift ; Inverted ; Master ; Source ; Targets ;
+------------+------+--------+-----------+-------+-------+------------+-----------+-------------+-------+--------+----
-------+------------+----------+--------+--------+---------+
; clk        ; Base ; 10.000 ; 100.0 MHz ; 0.000 ; 5.000 ;            ;           ;             ;       ;        ;           ;
;            ;          ;        ;        ; { clk } ;
+------------+------+--------+-----------+-------+-------+------------+-----------+-------------+-------+--------+----
-------+------------+----------+--------+--------+---------+
+-------------------------------------------------+
; Slow 1100mV 0C Model Fmax Summary                ;
+------------+-----------------+------------+------+
; Fmax       ; Restricted Fmax ; Clock Name ; Note ;
+------------+-----------------+------------+------+
; 123.59 MHz ; 123.59 MHz      ; clk        ;      ;
+------------+-----------------+------------+------+
This panel reports FMAX for every clock in the design, regardless of the user-specified clock
periods.  FMAX is only computed for paths where the source and destination registers or ports
are driven by the same clock.  Paths of different clocks, including generated clocks, are ignored.
For paths between a clock and its inversion, FMAX is computed as if the rising and falling edges
are scaled along with FMAX, such that the duty cycle (in terms of a percentage) is maintained.
Altera recommends that you always use clock constraints and other slack reports for sign-off
analysis.



+-----------------------------------+
; TimeQuest Timing Analyzer Messages ;
+-----------------------------------+
Info: *******************************************************************
Info: Running Quartus II 32-bit TimeQuest Timing Analyzer
    Info: Version 13.1.4 Build 182 03/12/2014 SJ Full Version
    Info: Processing started: Sat Sep 12 21:21:54 2015
Info: Command: quartus_sta mp1 -c mp1
Info: qsta_default_script.tcl version: #1
Info (11104): Parallel Compilation has detected 24 hyper-threaded processors. However, the
extra hyper-threaded processors will not be used by default. Parallel Compilation will use 12 of
the 12 physical processors detected instead.
Info (21077): Core supply voltage is 1.1V
Info (21077): Low junction temperature is 0 degrees C
Info (21077): High junction temperature is 85 degrees C
Info (332104): Reading SDC File: 'mp1.out.sdc'
Info: Found TIMEQUEST_REPORT_SCRIPT_INCLUDE_DEFAULT_ANALYSIS = ON
Info: Analyzing Slow 1100mV 85C Model
Info (332146): Worst-case setup slack is 1.270
    Info (332119):     Slack       End Point TNS Clock
    Info (332119): ========= =================== =====================
    Info (332119):     1.270               0.000 clk
Info (332146): Worst-case hold slack is 0.297
    Info (332119):     Slack       End Point TNS Clock
    Info (332119): ========= =================== =====================
    Info (332119):     0.297               0.000 clk
Info (332140): No Recovery paths to report
Info (332140): No Removal paths to report
Info (332146): Worst-case minimum pulse width slack is 4.373
    Info (332119):     Slack       End Point TNS Clock
    Info (332119): ========= =================== =====================
    Info (332119):     4.373               0.000 clk
Info: Analyzing Slow 1100mV 0C Model
Info (334003): Started post-fitting delay annotation
Info (334004): Delay annotation completed successfully
Info (332146): Worst-case setup slack is 1.909
    Info (332119):     Slack       End Point TNS Clock
    Info (332119): ========= =================== =====================
    Info (332119):     1.909               0.000 clk
Info (332146): Worst-case hold slack is 0.276
    Info (332119):     Slack       End Point TNS Clock
    Info (332119): ========= =================== =====================
    Info (332119):     0.276               0.000 clk
Info (332140): No Recovery paths to report
Info (332140): No Removal paths to report
Info (332146): Worst-case minimum pulse width slack is 4.373
    Info (332119):     Slack       End Point TNS Clock
    Info (332119): ========= =================== =====================
    Info (332119):     4.373               0.000 clk
Info: Analyzing Fast 1100mV 0C Model
Info (332146): Worst-case setup slack is 4.698
    Info (332119):     Slack       End Point TNS Clock
    Info (332119): ========= =================== =====================
    Info (332119):     4.698               0.000 clk
Info (332146): Worst-case hold slack is 0.184
    Info (332119):     Slack       End Point TNS Clock
    Info (332119): ========= =================== =====================
    Info (332119):     0.184               0.000 clk
Info (332140): No Recovery paths to report
Info (332140): No Removal paths to report
Info (332146): Worst-case minimum pulse width slack is 4.658
    Info (332119):     Slack       End Point TNS Clock
    Info (332119): ========= =================== =====================
    Info (332119):     4.658               0.000 clk
Info (332102): Design is not fully constrained for setup requirements
Info (332102): Design is not fully constrained for hold requirements
Info: Quartus II 32-bit TimeQuest Timing Analyzer was successful. 0 errors, 0 warnings
    Info: Peak virtual memory: 576 megabytes
    Info: Processing ended: Sat Sep 12 21:22:04 2015
```

```
Info: Elapsed time: 00:00:10
Info: Total CPU time (on all processors): 00:00:09
```