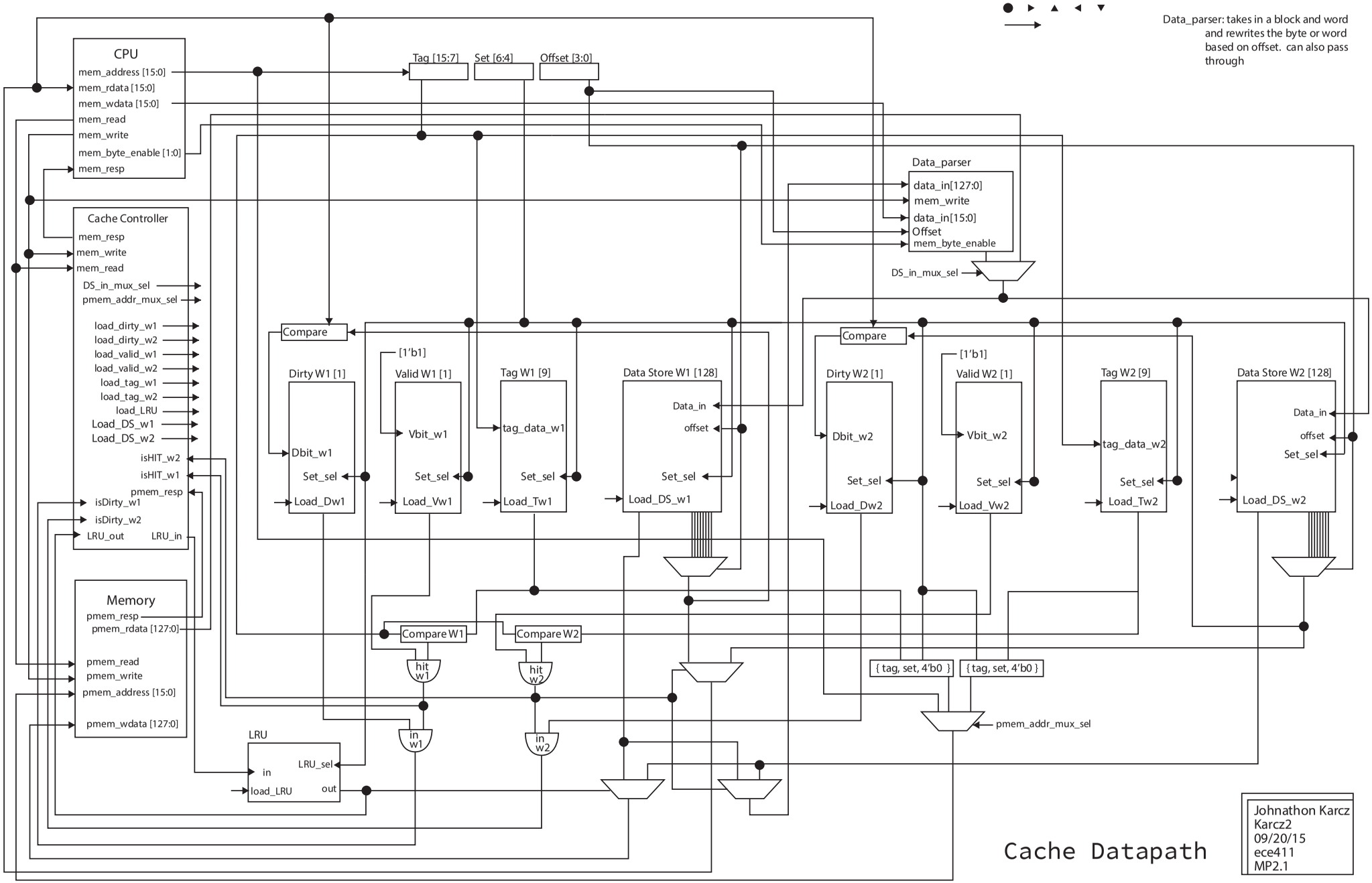
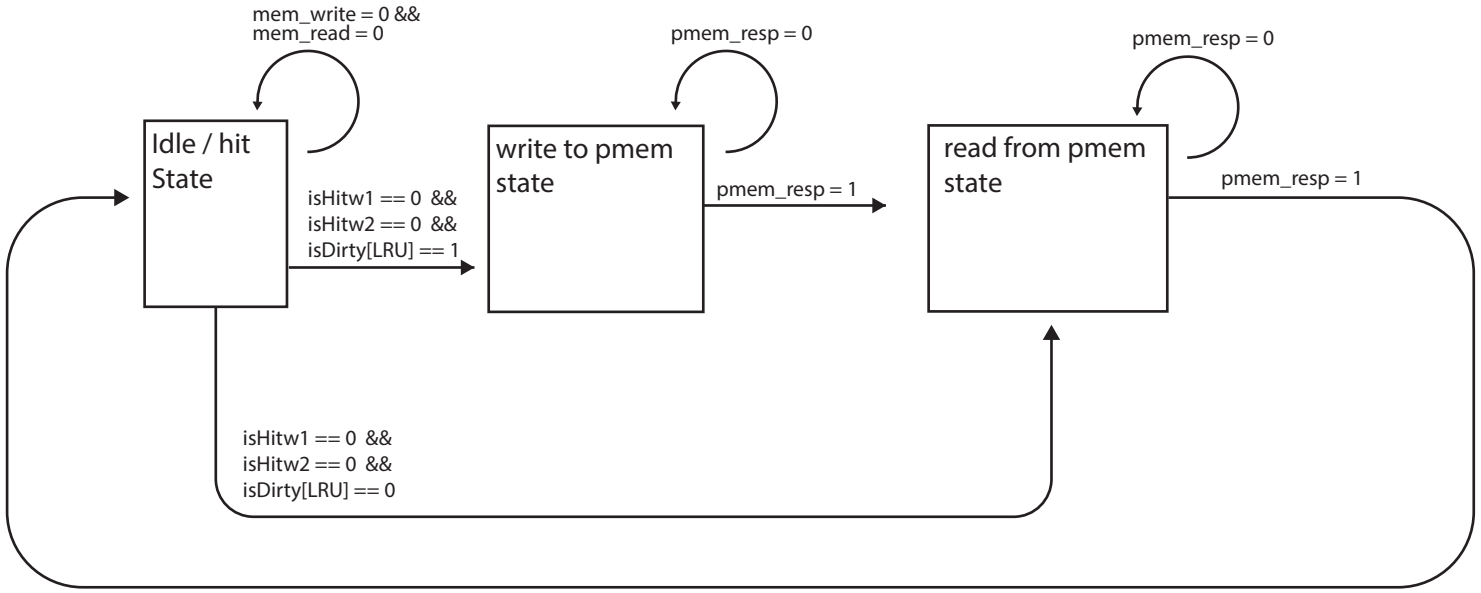
Johnathon Karcz
NetID: karcz2

ECE411
MP2
Final Hand In

10/03/2015

Paper Design of your Datapath


Data_parser: takes in a block and word and rewrites the byte or word based on offset. can also pass through
CPU
mem_address [15:0]
mem_rdata [15:0]
mem_wdata [15:0]
mem_read
mem_write
mem_byte_enable [1:0]
mem_resp
Tag [15:7]
Set [6:4]
Offset [3:0]
Data_parser
data_in[127:0]
mem_write
data_in[15:0]
Offset
mem_byte_enable
DS_in_mux_sel
Cache Controller
mem_resp
mem_write
mem_read
DS_in_mux_sel
pmem_addr_mux_sel
load_dirty_w1
load_dirty_w2
load_valid_w1
load_valid_w2
load_tag_w1
load_tag_w2
load_LRU
Load_DS_w1
Load_DS_w2
isHIT_w2
isHIT_w1
pmem_resp
isDirty_w1
isDirty_w2
LRU_out
LRU_in
Compare
[1'b1]
Dirty W1 [1]
Valid W1 [1]
Tag W1 [9]
Data Store W1 [128]
Dbit_w1
Vbit_w1
tag_data_w1
Data_in
offset
Set_sel
Load_Dw1
Load_Vw1
Load_Tw1
Load_DS_w1
Compare
[1'b1]
Dirty W2 [1]
Valid W2 [1]
Tag W2 [9]
Data Store W2 [128]
Dbit_w2
Vbit_w2
tag_data_w2
Data_in
offset
Set_sel
Load_Dw2
Load_Vw2
Load_Tw2
Load_DS_w2
Memory
pmem_resp
pmem_rdata [127:0]
pmem_read
pmem_write
pmem_address [15:0]
pmem_wdata [127:0]
Compare W1
Compare W2
hit w1
hit w2
{ tag, set, 4'b0 }
{ tag, set, 4'b0 }
pmem_addr_mux_sel
in w1
in w2
LRU
in
load_LRU
LRU_sel
out
Cache Datapath
Johnathon Karcz
Karcz2
09/20/15
ece411
MP2.1



Cache Control

Johnathon Karcz
Karcz2
10/03/15
ece411
MP2 final

Description of Cache Datapath Blocks

WAY BLOCKS

[Each block in this section is instantiated twice, once for each way]

X = way number

dirty_compare_wX

Not really needed based on piazza post @217, but I had it in there nonetheless, so I thought i would mention its function. This module takes in the word that is to be written and the current word in the cache for the set and offset, and checks if they match. If they do, then we do NOT set the dirty bit. The idea here is that if they match, even though we did a write, the data in cache and data in memory still match. Therefore the cache is not dirty. This check could save unnecessary write backs.

dirty_array_wX

This is the array that holds the dirty bits for each block, 8 in total per way. Inputs are a load signal, the set index, and a dirty bit in. This in value comes from the controller, so we have control of setting to 1 for write and 0 for a replacement of data without write. The output is the value that is in the array for a given index. The same index is used for input (writing) and output. The dirty bit for a given block is set when we write to the cache. With the exception if both the wdata and data in cache. See dirty_compare_wX above.

valid_array_wX

Similar to the dirty array, this module will hold 8 values, one for each block. Keeping track of cache lines that actually hold valid data. Any time we bring data from memory into the cache, we will set the valid bit to 1 for that specific set block. In contrast to the dirty array, the input for the valid array is hard set to VCC. At least for this cache, data is never removed and cleared, once loaded, even if replaced, the data is always valid. Just as with the dirty array, the output is the data stored at a given index of the array.

tag_array_wX

Inputs here are again the set index and a write signal. Again, the main difference being the data in/out. The data in is the nine most significant bits of the incoming mem_address from the CPU. This is stored as our unique comparator for data blocks.

datastore_array_wX

Most importantly we have the data store array. I am glad this is the last array, so i can finally stop saying that this array also takes an input of set index and write signal from the controller

unit. The input / output data signals are again unique here. Clearly we are taking in the data to be stored, but in large chunks. For our cache, we are storing 128 bit blocks, one block for each set. If we want to write to a single word in the cache this needs to happen outside of the array. So we output the data for that set, send it to another block, write, and then write the altered block back into the same index of the cache.

datastore_out_mux_wX

As mentioned above the data store array holds 128 bits of data for each set. Therefore if we wanted to send one of those words back to the CPU for loading into a register, for example, we need a way parse out the words from the entire block. this is where the data store out mux comes in. This mux has 8 inputs one for each sequential word in the block.

Example: block[127:112], block[111:96], …, block[15:0]

The select bit for the output of the mux is the offset bits of the in incoming mem_address from the CPU. The result is that we are only outputting the word that is selected by the offset of the mem_address from the CPU.

dirty_hit_wX

This module is basically just a set of comparators. I made it onto its own module to simplify things and keep them all in one place. Inputs are the tag from the mem_address, and the tag of the current way and set. This module also takes the in the outputs of the dirty and valid arrays. Once inside there is a comparison of tags to check for hits, this is ANDed with the valid bit as well, then the signal is outputted as “hit_out”. There is also logic that ANDs together the valid and dirty bit to check for a proper dirty bit, this signal is sent out as “dirty_out”. arguably you really do not need to AND the dirty and valid bits together because there should be no way for the dirty bit to be set without the valid bit also being set. Unfortunately, this is an artifact of additional logic that I *thought* I needed earlier in the design process. There would have been a lot of refactoring needed to fix it, and in terms of area, it is only adding two AND gates. I decided to leave it.

SHARED BLOCKS

[These blocks are shared between the two ways within the datapath]

data_way_mux_16

Two input mux, each being the outputs of the two datastore_out_muxes. Noted above the output of each of those muxes is 16 bits word selected by the offset. The select for this unit is the “isHit_w2” signal, meaning that if there is a hit on way 2 (1, 2 count scheme), then we want to output datastore 2 data. The output of this mux goes to the mem_rdata back the the CPU to be loaded into MDR.

data_way_mux_128

I would like to note that there are two muxes that take the full 128 bit output of each datastore and send them to various locations. The reason for having two of these muxes is that the input destination is not the same for the two muxes, which clearly could just send the data to two different locations, but the problem is that we want to use a different select signal depending on the receiving input. Noted below is the select signals output differences for each.

data_way_mux_128_parser

For the parser version of the mux, we will use the isHit_w2 signal, meaning that the second way has a hit. Otherwise we will have a hit from the first way. This will take the 128 bit output of the way that has the hit and send it to our Data Parser module (see below for description), so that we can write a new word or byte to the data currently in the array.

data_way_mux_128_pmem

For the pmem version of the 128 mux, we use the LRU output to select the way that we would like to select the data from. The output of the mux goes to the input of the physical memory. The idea here is that this mux is used when we need to evict data from the cache and the dirty bit for that block is high. The LRU will tell us which datastore to send to pmem, so that we can overwrite it in the next state.

lru

Another array, the LRU also uses the set bits in order to select the set that we are working in. The array data holds either a 1 or 0 depending on the way that has been least recently used for that set. The input for the LRU comes

datastore_in_mux

The purpose of the datastore in mux is to select between the two possible inputs for either of the two data store arrays. This takes the full 128 bit input from the datastore parser, and the input from pmem_rdata. The control unit executes the select bit. The idea is to be able to select pmem as the input to the data store when writing memory to the cache. Alternatively, we are also able to re-write the date that comes from the cache, but with the new value written in by the data parser.

pmem_address_mux

The address that is send to pmem_address can come from one of two sources. Either the cache or the original mem_address. Once a block of memory is written into the cache, the address from which it originated is preserved via the tag and set bits. So, as a result this is a 4 input mux (only 3 spots are used), where we set two of the inputs for mux to be a concatenation of the mentioned signals in order to reconstruct the signal, like so:
{tag_array_w1_out, set, 4'b0}

{tag_array_w2_out, set, 4'b0}
The third (and fourth) inputs are simply the full, original mem_address signal. The select signal is outputted from the control unit and selects the appropriate output based on the state.

datastore_parser

Finally, we have the often talked about data parser. Technically the Data Parser unit is a series of muxes and a decoder. It takes a 128 bit input of the data that can be found in the data store array (selectable by the data_way_mux_128 listed above). It also takes in the signals mem_wdata, mem_byte_enable, offset, and mem_write from the CPU. The output is the original 128 signal sent in from the data store, except with either a word or byte from the mem_wdata signal written into the 128 bits based on the offset and mem_byte_enable signals. Just as before we had the cache, we need to be able to write using STB a byte, we are still able to do that with the data parser. The output goes to the datastore in mux, as noted above.

Explain How You Tested Your Design

In order to test my cache, I had to take a different approach from that of MP1. In this case much of the interaction was with the cache and memory, and there were very specific sets of instructions that were needed in order to test the operations of the unit.

I started by filling out blocks of memory that i can see and call as i need them. I then proceeded to make sure that i can fill the entire cache without a problem. It was difficult to predict in full exactly what would happen when it came to hits and stores, because the instructions and blocks that contain labels also need to be loaded into the cache. So there were times when you would get to the point where Fetch2 would miss, and you would need to allow fetch 2 to read from pmem and load into the cache. As a solution to this, I would trace along, instruction by instruction and make sure that the signals that i needed at any given time were being correctly set. If I noticed a new line coming in because of the fetch, i would load the memory list and see where the instruction lies in memory, and make sure that the load is valid.

After confirming that a specific action worked the way i wanted I would add another test to my code given the state of the machine and things that I could make happen on the next instruction. For example, if I had a block replace, then store to that block. I might execute two loads one after each other that target the same exact set in order to force an eviction of that cache block that was written to. Then I would check the mem list to make sure that the memory was written to on the evict correctly.

Another thing that I did in my tests was ran specific tests that executed all of the load and store instructions found in the ISA. My concern was that the cache adds a considerable layer to all of the load and store instructions, and the all needed to be retested to make sure that each of their unique functionalities still worked.

Basically there were a lot of signals to keep track of and a lot of moving parts. I would keep adding instructions checking all of the relevant signals and making sure that there was no operation of the cache being ignored. I feel like I was able to get complete coverage with my testing. Only at that point did i run the test code. I was pleased to see that i was able to get the same register values with my cpu and cache, as found in the lc3 simulator.

Below is one of the test code files that i had written for testing.

Personal Test Code

```
ORIGIN 0
SEGMENT 0 CODE:
;; instruction will be loaded into w0s0
      LEA R1, data
```

```
     LDR R2, R1, l2p ; R2 points to the head of segment line2
     LDR R3, R1, l3p ; R3 points to the head of segment line3
;; load up all of the cache to make sure that all the arrays
;;  are being set correctly.
     LDR R4, R2, A10 ; load:s1
     LDR R4, R2, A20 ; load:s2
     LDR R4, R2, A30 ; load:s3
     LDR R4, R3, A40 ; load:s4
     LDR R4, R3, A50 ; load:s5
     LDR R4, R3, A60 ; load:s6
     LDR R7, R3, A70 ; load:s7

     LDR R4, R7, B00 ; load:s0
     LDR R4, R7, B10 ; load:s1
     LDR R4, R7, B20 ; load:s2
     LDR R7, R7, B30 ; load:s3
     LDR R4, R7, B40 ; load:s4
     LDR R4, R7, A50 ; load:s5
     LDR R4, R7, A60 ; load:s6
     LDR R4, R7, A70 ; load:s7
;; Just to help find out spot
     ADD R7, R0, -16
;; test to make sure that store works
     LDR R4, R1, good
     LEA r2, line0A
     ADD r3, r2, 1
     STR r4, r2, A02
     STR r4, r2, A04
;; test to make sure stb works
     STB r4, r2, A06
     RSHFL r4, r4, 8
     STB r4, r3, A06
;; test to make sure that LDR works
     lea r7, line0C
     ldr r7, r7, C00
     ldr r7, r7, B00
;; test to make sure that sti works
     lea r5, data
     add r6, r5, 1
     ldr r4, r5, cool
     sti r4, r7, B40
;; test to make sure that ldb works
     ldb r4, r5, count
```

```
     ldb r4, r6, count
;; test to make sure that ldi works
     ldi r4, r5, boob
     ldi r3, r5, fin
     ldr r3, r3, C05
;; finish in loop
inf:
     BRnzp inf

SEGMENT 96 data:
;l1p:DATA2 leetline
l2p:  DATA2 line0A
l3p:  DATA2 line4A
fin:  DATA2 A07
badd:      DATA2 4xBADD
bad:  DATA2 4xBAD
good:      DATA2 4x600D
god:  DATA2 4x060D
count:     DATA2 4x1234
cool:      DATA2 4xC001
dude:      DATA2 4xD00D
boob:DATA2 A14

SEGMENT 32 line0A:
A00:  DATA2 line0B ; 4xA0A0
A01:  DATA2 4xA0A1
A02:  DATA2 4xA0A2
A03:  DATA2 4xA0A3
A04:  DATA2 4xA0A4
A05:  DATA2 4xA0A5
A06:  DATA2 4xA0A6
A07:  DATA2 line0C
;SEGMENT 16 line1A:
A10:  DATA2 4xA1A0
A11:  DATA2 4xA1A1
A12:  DATA2 4xA1A2
A13:  DATA2 4xA1A3
A14:  DATA2 4xB00B
A15:  DATA2 4xA1A5
A16:  DATA2 4xA1A6
A17:  DATA2 4xA1A7
;SEGMENT 16 line2A:
A20:  DATA2 4xA2A0
```

```
A21:  DATA2 4xA2A1
A22:  DATA2 4xA2A2
A23:  DATA2 4xA2A3
A24:  DATA2 4xA2A4
A25:  DATA2 4xA2A5
A26:  DATA2 4xA2A6
A27:  DATA2 4xA2A7
;SEGMENT 16 line3A:
A30:  DATA2 4xA3A0
A31:  DATA2 4xA3A1
A32:  DATA2 4xA3A2
A33:  DATA2 4xA3A3
A34:  DATA2 4xA3A4
A35:  DATA2 4xA3A5
A36:  DATA2 4xA3A6
A37:  DATA2 4xA3A7
SEGMENT 64 line4A:
A40:  DATA2 4xA4A0
A41:  DATA2 4xA4A1
A42:  DATA2 4xA4A2
A43:  DATA2 4xA4A3
A44:  DATA2 4xA4A4
A45:  DATA2 4xA4A5
A46:  DATA2 4xA4A6
A47:  DATA2 4xA4A7
;SEGMENT 16 line5A:
A50:  DATA2 4xA5A0
A51:  DATA2 4xA5A1
A52:  DATA2 4xA5A2
A53:  DATA2 4xA5A3
A54:  DATA2 4xA5A4
A55:  DATA2 4xA5A5
A56:  DATA2 4xA5A6
A57:  DATA2 4xA5A7
;SEGMENT 16 line6A:
A60:  DATA2 4xA6A0
A61:  DATA2 4xA6A1
A62:  DATA2 4xA6A2
A63:  DATA2 4xA6A3
A64:  DATA2 4xA6A4
A65:  DATA2 4xA6A5
A66:  DATA2 4xA6A6
A67:  DATA2 4xA6A7
```

```
;SEGMENT 16 line7A:
A70:  DATA2 line0B ; 4xA7A0
A71:  DATA2 4xA7A1
A72:  DATA2 4xA7A2
A73:  DATA2 4xA7A3
A74:  DATA2 4xA7A4
A75:  DATA2 4xA7A5
A76:  DATA2 4xA7A6
A77:  DATA2 4xA7A7

SEGMENT 64 line0B:
B00:  DATA2 line4B
B01:  DATA2 4xB0B1
B02:  DATA2 4xB0B2
B03:  DATA2 4xB0B3
B04:  DATA2 4xB0B4
B05:  DATA2 4xB0B5
B06:  DATA2 4xB0B6
B07:  DATA2 4xB0B7
;SEGMENT 16 line1B:
B10:  DATA2 4xB1B0
B11:  DATA2 4xB1B1
B12:  DATA2 4xB1B2
B13:  DATA2 4xB1B3
B14:  DATA2 4xB1B4
B15:  DATA2 4xB1B5
B16:  DATA2 4xB1B6
B17:  DATA2 4xB1B7
;SEGMENT 16 line2B:
B20:  DATA2 4xB2B0
B21:  DATA2 4xB2B1
B22:  DATA2 4xB2B2
B23:  DATA2 4xB2B3
B24:  DATA2 4xB2B4
B25:  DATA2 4xB2B5
B26:  DATA2 4xB2B6
B27:  DATA2 4xB2B7
;SEGMENT 16 line3B:
B30:  DATA2 line4B;4xB3B0
B31:  DATA2 4xB3B1
B32:  DATA2 4xB3B2
B33:  DATA2 4xB3B3
B34:  DATA2 4xB3B4
```

```
B35:  DATA2 4xB3B5
B36:  DATA2 4xB3B6
B37:  DATA2 4xB3B7
SEGMENT 64 line4B:
B40:  DATA2 C03
B41:  DATA2 4xB4B1
B42:  DATA2 4xB4B2
B43:  DATA2 4xB4B3
B44:  DATA2 4xB4B4
B45:  DATA2 4xB4B5
B46:  DATA2 4xB4B6
B47:  DATA2 4xB4B7
;SEGMENT 16 line5B:
B50:  DATA2 4xB5B0
B51:  DATA2 4xB5B1
B52:  DATA2 4xB5B2
B53:  DATA2 4xB5B3
B54:  DATA2 4xB5B4
B55:  DATA2 4xB5B5
B56:  DATA2 4xB5B6
B57:  DATA2 4xB5B7
;SEGMENT 16 line6B:
B60:  DATA2 4xB6B0
B61:  DATA2 4xB6B1
B62:  DATA2 4xB6B2
B63:  DATA2 4xB6B3
B64:  DATA2 4xB6B4
B65:  DATA2 4xB6B5
B66:  DATA2 4xB6B6
B67:  DATA2 4xB6B7
;SEGMENT 16 line7B:
B70:  DATA2 4xB7B0
B71:  DATA2 4xB7B1
B72:  DATA2 4xB7B2
B73:  DATA2 4xB7B3
B74:  DATA2 4xB7B4
B75:  DATA2 4xB7B5
B76:  DATA2 4xB7B6
B77:  DATA2 4xB7B7
SEGMENT 64 line0C:
C00:  DATA2 line0B
C01:  DATA2 4xC0C1
C02:  DATA2 4xD00D
```

```
C03: DATA2 4xBADD
C04: DATA2 4xC0C4
C05: DATA2 4xDEAD
C06: DATA2 4xC0C6
C07: DATA2 4xC0C7
```

Wave Traces index

Page1: Initial wave Trace

Length: 4500ns

Signals:

1. data store for way 0 (expanded) in hex
2. tag store for way 0 (expanded) in hex
3. pc_out

Page2: Initial wave Trace

Length: 4500ns

Signals:

1. data store for way 1 (expanded) in hex
2. tag store for way 1 (expanded) in hex
3. pc_out

Page3: Initial wave Trace

Length: 4500ns

Signals:

1. clk,
2. pc_out,
3. mem_address,
4. mem_read,
5. mem_rdata,
6. mem_write,
7. mem_byte_enable,
8. mem_wdata
9. registers from register file (expanded)
10. datapath’s current state

Page4: End of test code

Length: 1000ns

Signals:

1. data store for way 0 (expanded) in hex
2. tag store for way 0 (expanded) in hex
3. pc_out

Page5: End of test code

Length: 1000ns

Signals:

1. data store for way 1 (expanded) in hex
2. tag store for way 1 (expanded) in hex
3. pc_out

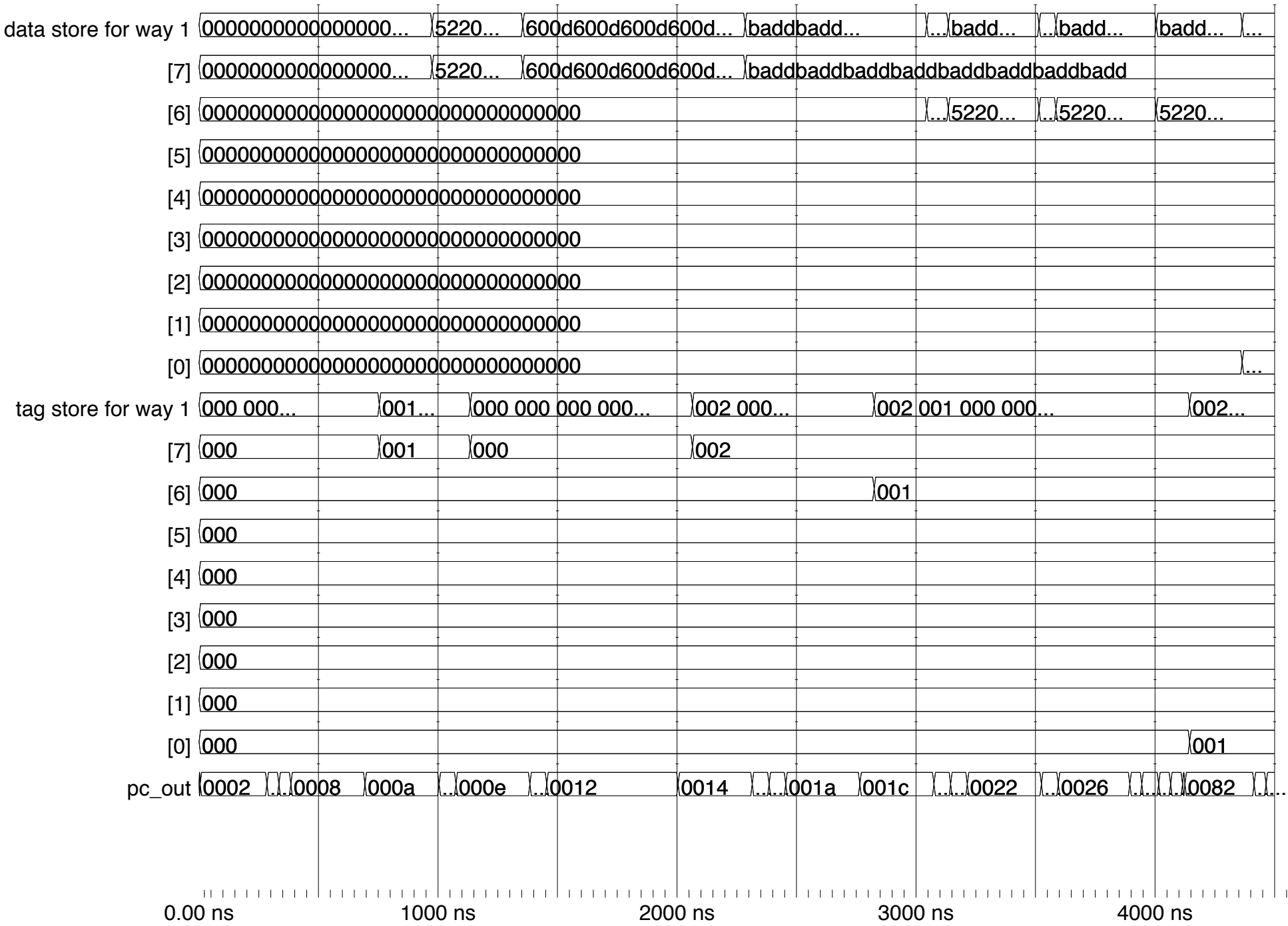
Page6: End of test code

Length: 1000ns

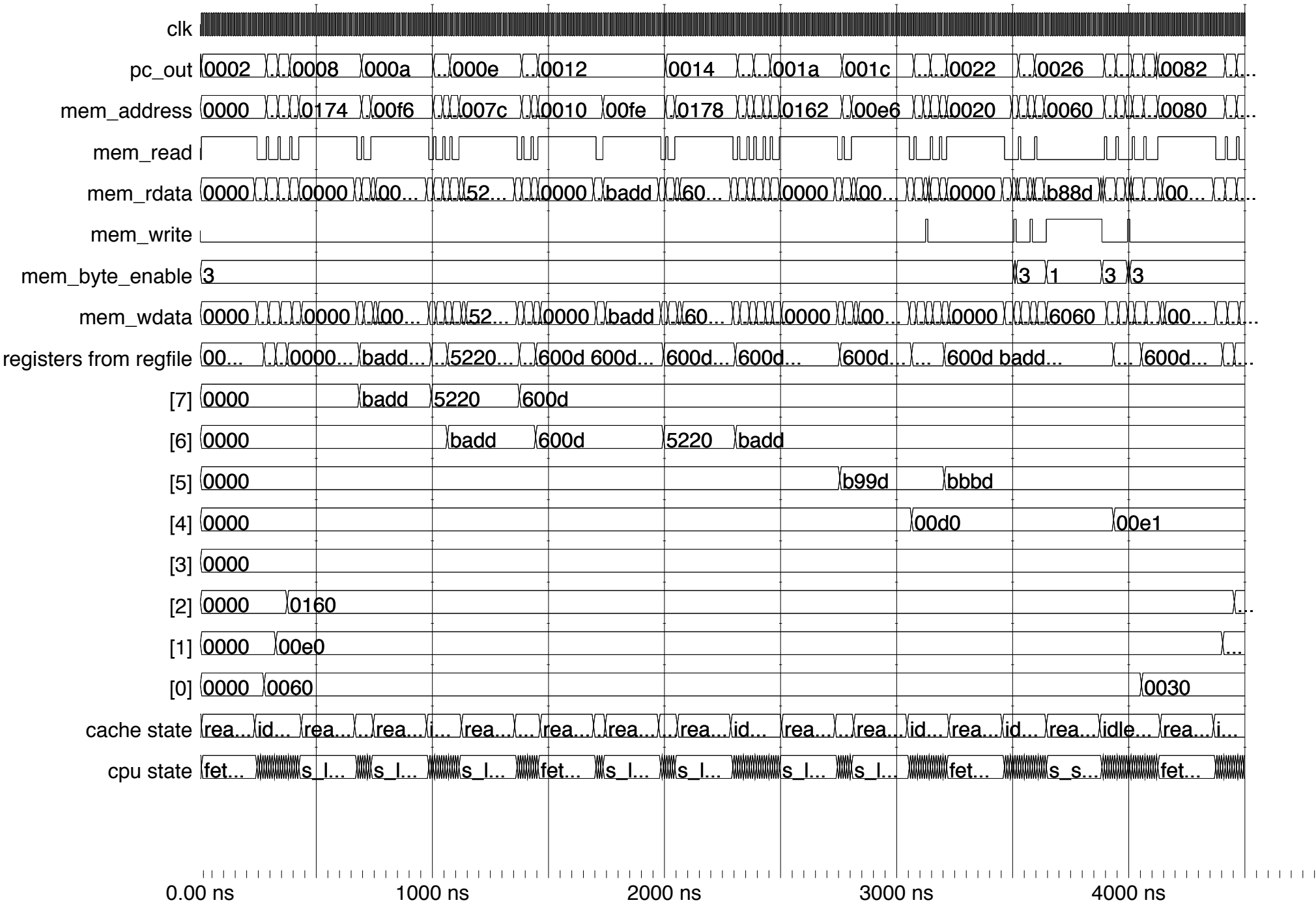
Signals:

1. clk,
2. pc_out,
3. mem_address,
4. mem_read,
5. mem_rdata,
6. mem_write,
7. mem_byte_enable,
8. mem_wdata
9. registers from register file (expanded)
10. datapath’s current state

Entity:mp2_tb Architecture: Date: Thu Oct 01 23:18:17 CDT 2015 Row: 1 Page: 1



Entity:mp2_tb Architecture: Date: Thu Oct 01 23:25:21 CDT 2015 Row: 1 Page: 1



Entity:mp2_tb Architecture: Date: Thu Oct 01 23:33:45 CDT 2015 Row: 1 Page: 1

Entity:mp2_tb Architecture: Date: Thu Oct 01 23:22:21 CDT 2015 Row: 1 Page: 1

Entity:mp2_tb Architecture: Date: Thu Oct 01 23:25:38 CDT 2015 Row: 1 Page: 1

Entity:mp2_tb Architecture: Date: Thu Oct 01 23:33:26 CDT 2015 Row: 1 Page: 1

list of all memory writes

ps delta		/mp2_tb/pmem_wdata	/mp2_tb/pmem_address /mp2_tb/pmem_write
0	+0	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxx x
0	+1	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx	xxxx 0
5115000	+5	600d600d600d600d600d600d600d0030	0050 1
5345000	+5	600d600d600d600d600d600d600d0030	0050 0
5645000	+5	5220522052205220522052200b05220	0150 1
5875000	+5	5220522052205220522052200b05220	00d0 0
6175000	+6	baadbaadbaadbaadbaad0130baadbaad	0150 1
6405000	+5	baadbaadbaadbaadbaad0130baadbaad	0150 0
6715000	+6	600d600d600d0130600d600d600d0030	00d0 1
6945000	+5	600d600d600d0130600d600d600d0030	0050 0
11535000	+6	600d600d0060600d600d600d600d60fa	0030 1
11765000	+5	600d600d0060600d600d600d600d60fa	0030 0

Timing Analysis Report

+--+
; TimeQuest Timing Analyzer Summary ;
+--------------------+---+
; Quartus II Version ; Version 13.1.4 Build 182 03/12/2014 SJ Full Version ;
; Revision Name ; mp2 ;
; 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 85C Model Fmax Summary ;
+------------+----------------+------------+------+
; Fmax ; Restricted Fmax ; Clock Name ; Note ;
+------------+----------------+------------+------+
; 105.73 MHz ; 105.73 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: Thu Oct  1 20:07:28 2015
Info: Command: quartus_sta mp2 -c mp2
Info: qsta_default_script.tcl version: #1
Info (20030): Parallel compilation is enabled and will use 4 of the 4 processors detected
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: 'mp2.out.sdc'
Info: Found TIMEQUEST_REPORT_SCRIPT_INCLUDE_DEFAULT_ANALYSIS = ON
Info: Analyzing Slow 1100mV 85C Model
Info (332146): Worst-case setup slack is 0.542
    Info (332119):     Slack      End Point TNS Clock
    Info (332119): ========= ==================== ====================
    Info (332119):     0.542             0.000 clk
Info (332146): Worst-case hold slack is 0.301
    Info (332119):     Slack      End Point TNS Clock
    Info (332119): ========= ==================== ====================
    Info (332119):     0.301             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 4.370
    Info (332119):     Slack      End Point TNS Clock
    Info (332119): ========= ==================== ====================
    Info (332119):     4.370             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 1.223
```

Info (332119): Slack End Point TNS Clock
Info (332119): ========= ==================== ====================
Info (332119): 1.223 0.000 clk
Info (332146): Worst-case hold slack is 0.277
Info (332119): Slack End Point TNS Clock
Info (332119): ========= ==================== ====================
Info (332119): 0.277 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 4.372
Info (332119): Slack End Point TNS Clock
Info (332119): ========= ==================== ====================
Info (332119): 4.372 0.000 clk
Info: Analyzing Fast 1100mV 0C Model
Info (332146): Worst-case setup slack is 3.768
Info (332119): Slack End Point TNS Clock
Info (332119): ========= ==================== ====================
Info (332119): 3.768 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 4.654
Info (332119): Slack End Point TNS Clock
Info (332119): ========= ==================== ====================
Info (332119): 4.654 0.000 clk
Info (332101): Design is fully constrained for setup requirements
Info (332101): Design is fully constrained for hold requirements
Info: Quartus II 32-bit TimeQuest Timing Analyzer was successful. 0 errors, 0 warnings
Info: Peak virtual memory: 536 megabytes
Info: Processing ended: Thu Oct 1 20:07:46 2015
Info: Elapsed time: 00:00:18
Info: Total CPU time (on all processors): 00:00:06