Final Project

ENEE459D (0101) w/ Pramod Govindan

Team Members:
Richard Ojo
Noel Hasson

12/18/2020

Table of Contents

II. Signature Approval

My contribution to the projects consists of responsibility for the simple processor, the APB master, the related individual and system testbenches, collaborating with my partner on the general outline for the system testbench code, and participation in the mid-term and final presentation and report efforts.

I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/ examination.

Richard Ojo

My contribution to the project consists of responsibility for the watchdog peripheral and the related individual and system testbenches, collaborating with my partner on the general outline for the system testbench code, and participation in the mid-term and final presentation and report efforts.

I pledge on my honor that I have not given or received any unauthorized assistance on this assignment/examination.

Noel Hasson

III. Executive Summary

This report describes the design, development, and verification of a small System-on-Chip (SoC) environment which utilizes the Advanced Peripheral Bus (APB) interface. The APB is part of the Advanced Microcontroller Bus Architecture (AMBA) protocol family. AMBA defines how to manage and efficiently connect SoC components. The components covered in this paper will include a CPU, two peripherals - a memory (RAM) module, and a watchdog timer (WDT) module - along with the required master and slave APB peripheral interfaces.

The path to achieve a successful implementation of this SoC environment will proceed in the following order: The CPU and master APB modules will be designed together, an external memory module will be used and designed with the APB slave module, and lastly the watchdog timer module will be designed. After design, the individual modules will be tested independently. The final step will be integrating all the components into a system level testbench making use of the standard stimulus generator, driver, and combined scoreboard/checker classes. Random stimulus testing and various directed testing at the system level will be used to root out bugs. All coding will use System Verilog and be done in the Xilinx Vivado software.

The CPU was designed using the basic von Neumann architecture [5], however, changes and updates were made to fit this project's end goals. The CPU utilizes four main registers: the program counter (PC), memory access register (MAR), memory data register (MDR), and the instruction register (IR). There is also a special register, memory peripheral prdata (MPPRDTA) register. This means that there is no accumulator (ACC) or arithmetic logic unit (ALU). This is because our design goals did not require any arithmetic computations. The CPU also has an internal instruction memory, where the CPU will read instructions from. The CPU is not pipelined and is thus a single instruction processor. This also allows the processor to have dedicated stages for particular instructions as a pipelined system requires the CPU to go through all stages for each instruction. The instructions that the CPU reads and executes are 32 bits, where the most significant three bits are always for the opcode.

The Watchdog unit is a useful APB slave module for monitoring a system's operation. The WDT module triggers or sends a reset signal to the processor in case of various system malfunctions. This should restore regular system functions relatively quickly thus preventing long term "locking up" or freezing. The design is influenced by the official arm (Advanced RISC Machine) watchdog timer module, but is much more simplistic. There are three values that can be written to the watchdog module by user instructions: a clock divider integer, a timeout value, and a "kick" value. The signal handling for these values are also treated in a simplistic manner: the cock divider and timeout values can be a 21-bit integer, while the kick is only expecting a high (1) signal for activation. This simplicity makes for a small SoC environment, it does mean the watchdog timer module is less robust in catching and handling system problems.

IV. Main Body

A. Introduction

This report describes the design, development, implementation, and verification of a peripheral subsystem with an Advanced Peripheral Bus (APB) interface. The APB is part of the Advanced Microcontroller Bus Architecture (AMBA) protocol family [2]. AMBA defines how to manage and efficiently connect System-on-Chip (SoC) components. These components include a CPU, Memory (RAM), storage, and a myriad of input/output (I/O) devices.

Specifically, the SoC peripheral detailed in this report is the Watchdog timer/unit (WDT). The Watchdog unit is a useful APB slave module for monitoring a system's operation. The WDT module triggers or sends a reset signal to the processor in case of system malfunctions. This should restore regular system functions, and thus prevents the system from completely "locking up" or freezing.

The final outcome is a completed WDT APB subsystem, which is connected to a processor. All modules are proven to be properly designed and fully functional using testbenches using SystemVerilog [3].

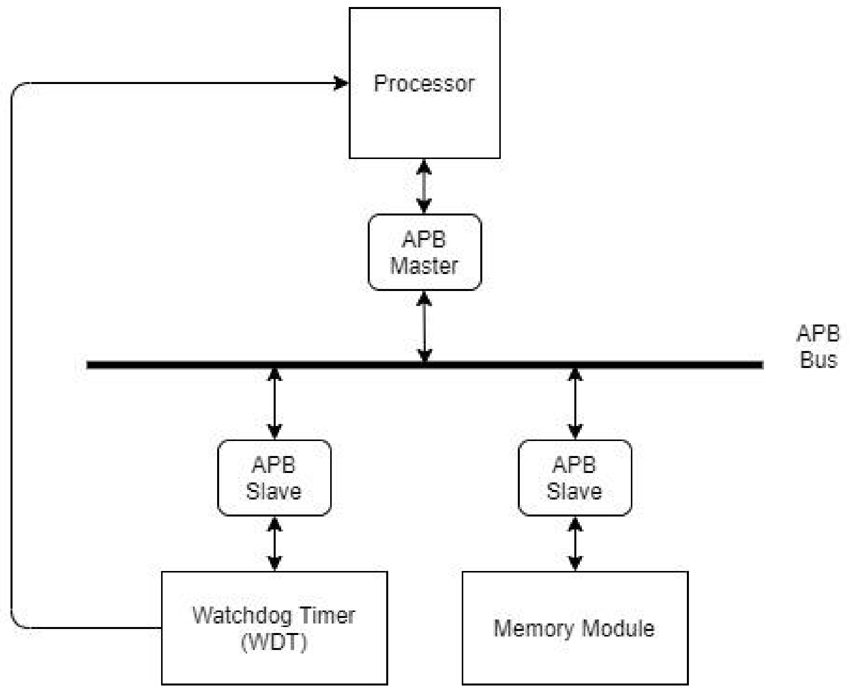


Fig. 1. System level diagram of the full APB subsystem with CPU

Design of the Watchdog Timer is heavily influenced by the ARM Watchdog Module Technical Reference Manual [4] but with many simplifications. The processor design was influenced by Intel's lab: A Simple Processor [5]. This type of design involving processor and peripherals communicating thru the APB bus is a simplified model of what could be implemented in an embedded device. A sophisticated Watchdog timer module is a critical component of spacecraft systems where human intervention when things go wrong is very difficult. A watchdog timer could be said to have saved the Pathfinder mission, whereas the Clementine spacecraft Geographos mission ended in failure due to the choice to not implement the hardware watchdog that was available, and instead use a software watchdog that didn't function properly when the system encountered a problem [6] .

B. Goals and Design Overview

Overall System

The goal of this project is to design a peripheral subsystem, along with a memory unit. The chosen peripheral is the Watchdog module, which is an AMBA slave module. This project also entails designing a simple processor with an APB master interface, which connects to the aforementioned Watchdog peripheral subsystem. All of the subsystem's components, the processor, and the memory module are implemented using SystemVerilog.

The processor and peripheral subsystem are tested and verified separately using testbenches created in SystemVerilog. Finally, the overall system's functionality is verified with a constrained pseudo testbench, which consists of a checker, a monitor, and a scoreboard.

Complete system design and verification is completed in eight weeks.

Central Processing Unit (CPU)

The CPU simply executes instructions that it is given. The CPU does this in multiple steps or stages. Each instruction goes through the first four steps: Fetch1, Fetch2, Decode, and Execute. During the Fetch process, the CPU grabs the next instruction from the instruction memory (at the location specified by the PC) and stores it in the CPU's instruction register (IR). During the Decode stage, the instruction's opcode is decoded (using a decoder) so that the CPU knows which instruction to execute. In the execute stage, the CPU drives signals to the rest of the system. The CPU asserts the CPUDONE signal once the instruction is finished.

TABLE I
CPU MODULES I/O SIGNALS

CPU Signal	**Function**	**Input/Output**	**Size (Bits)**
CCLK	Clock input	Input	1
CPURESET	Trigger a CPU reset	Input	1
RUN	Tells cpu to execute instructions	Input	1
CPUREADY	Signal sent from APB Master to alert the CPU that the instruction is complete	Input	1
INCPURESET	CPU Interrupt signal from WDT	Input	1
MPPRDATA	Data read from the memory peripheral for instruction 110	Input	21
APBMASTERENABLE	Enable the APB master/ tell the APB Master to run	Output	1
CPUSEL	Output signal that displays which instruction is being run	Output	8
CPUDONE	Signal that shows that the CPU has completed its instruction	Output	1
CPUPERPHRESET	Output signal to reset master and peripherals	Output	1
addr	Target address to either write or read from	Output	8
data	Data to write	Output	21

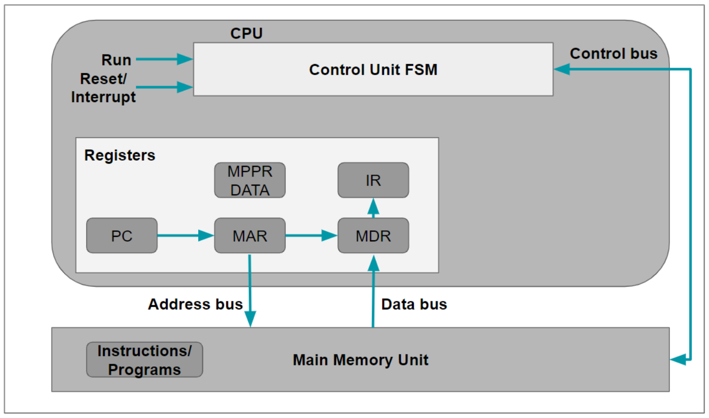


Fig. 2. High level diagram of the CPU Module

Instruction Set Architecture

Write:

3-bit Opcode	8-bit Address	21-bit Data

Read:

3-bit Opcode	8-bit Address	21-bit XXXXXXXXXXXXXXXXXXXXX

Instruction Memory Write:

3-bit Opcode	8-bit Address	21-bit XXXXXXXXXXXXXXXXXXXXX
32-bit New Instruction		

Memory read to WDT Write:

3-bit Opcode	8-bit WDT Write Addr	13-bit XXXXXXXXXXXXXXXXXXX	8-bit Memory Address

Fig. 3. The four instruction formats

The system has a 32 bit instruction set and the system uses an instruction set architecture (ISA) with four instruction formats (shown in the above image). The ISA uses 32 bit instructions, with a three bit opcode, and the remaining 29 bits vary based on the instruction. With a 3 bit opcode, the CPU has the ability to execute eight different instructions, but only 7 instructions are detailed in this report. An instruction with opcode “111 or 7” will stall the CPU and stall the subsystem as a result.
A list of instructions is given below:

TABLE II
CPU INSTRUCTIONS

Opcode	**Instruction**	**Function performed**
000	Reset System	Set to default values
001	Write to Watchdog peripheral	Set various: Timeout, clock divider, kick
010	Watchdog read	Read values from WDT
011	Instruction Memory Write	Imem[addr] = Data
100	Memory Module Write	mem[addr] = Data
101	Memory Module Read	PRDATA = mem[addr]
110	Memory Module Read to WDT Write	Timeout, clock divider, kick = mem[addr]

Instructions are loaded into the system's dedicated instruction/program memory module. When a manual Reset System occurs, the CPU's PC is reset to 0 and all registers are set to their default values. A System reset will also occur when the WDT sends the CPU an interrupt/reset signal.

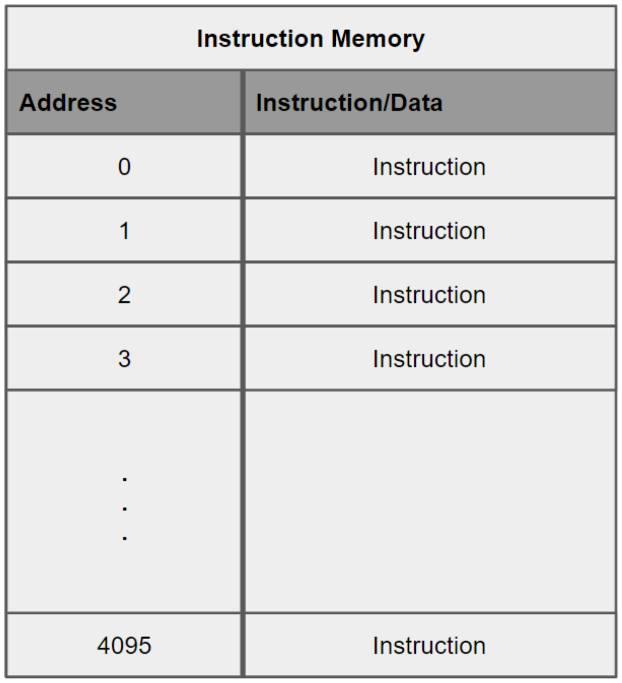
TABLE III
ADDITIONAL STAGES PER CPU INSTRUCTION

Opcode	**Instruction**	**Additional Stages**
000	Reset System	RESET
001	Write to Watchdog peripheral	IDLE
010	Watchdog read	IDLE
011	Instruction Memory Write	MEMORY1, MEMORY2, WRITEBACK1, WRITEBACK2
100	Memory Module Write	IDLE
101	Memory Module Read	IDLE
110	Memory Module Read to WDT Write	IDLE, MEMORY1

Different Instructions utilize more or different stages than the first four already mentioned (Fetch1, Fetch2, Decode, and Execute). The table above displays what those stages are.

Instruction Memory Module

The instruction memory module is contained inside the CPU module. The instruction memory stores the programs/instructions that the CPU will read and eventually execute. The instruction memory is 32 bits wide and 4KB deep. 4KB was chosen as the RAM size because this design would not need to execute a lot of instructions in memory. A smaller RAM size also conserves system resources.



Instruction Memory	
Address	**Instruction/Data**
0	Instruction
1	Instruction
2	Instruction
3	Instruction
. . .	
4095	Instruction

Fig. 3. Simple diagram of the instruction memory Block RAM

The CPU can also simulate a write back to the instruction memory. However, since the instruction needs to be in the instruction memory already it is more akin to a “move instruction” instruction.

APB Master

The APB master drives the address and control signals to the APB slaves. The master does this by decoding the opcode and then driving the correct PSELECT. This system utilizes an APB master module that follows basic APB protocols, and thus this module can be incorporated into other systems utilizing the APB standard.

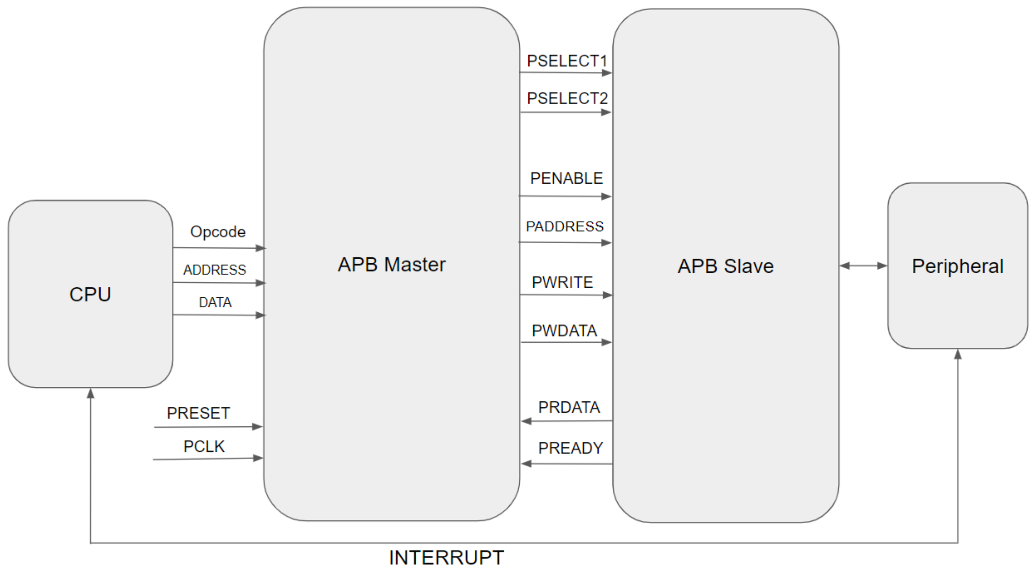


Fig. 4. High level diagram of the signals being passed between the CPU, APB Master, APB Slave, and Peripheral (WDT)

Watchdog Timer

A basic single stage Watchdog Timer implementation [7] was chosen. Upon reaching the timeout value the WDT will trigger a reset signal to the CPU. The internals of the module is seen in [Fig. 5] below:

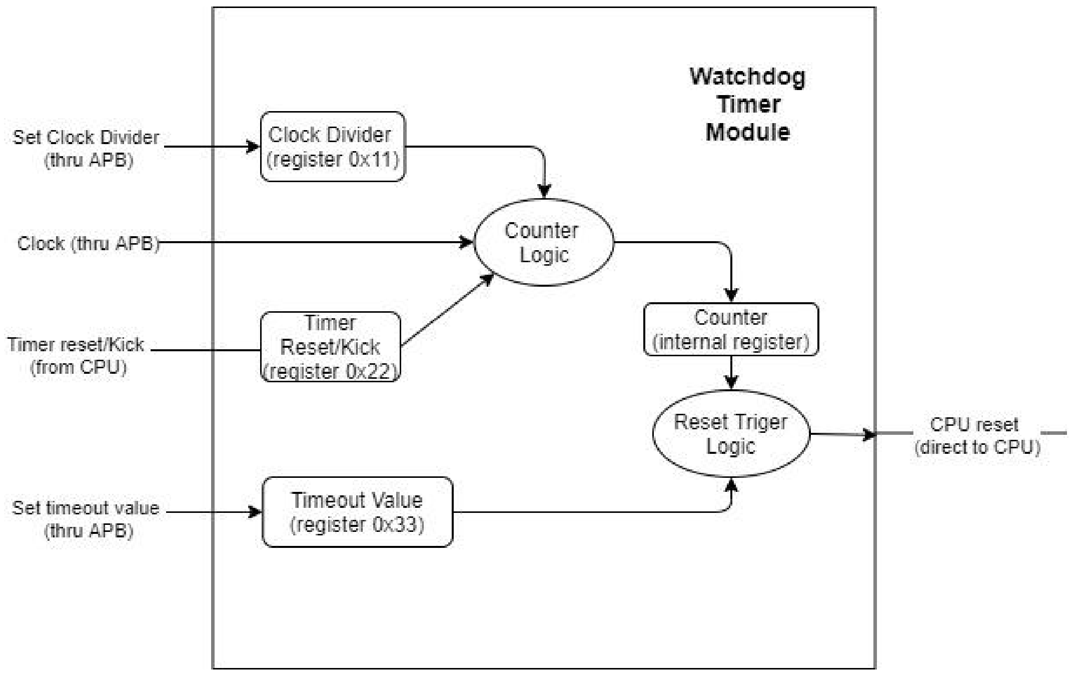

Watchdog Timer Module
Set Clock Divider (thru APB)
Clock Divider (register 0x11)
Clock (thru APB)
Counter Logic
Timer reset/Kick (from CPU)
Timer Reset/Kick (register 0x22)
Counter (internal register)
Reset Triger Logic
CPU reset (direct to CPU)
Set timeout value (thru APB)
Timeout Value (register 0x33)

Fig. 5. WatchDog Timer module

A brief description of the signals used is seen below.

- Clock Divider (8bit addr, 21bit data)
    - Increases WDT clock period by a multiple such that many clock cycles/instructions are run before WDT counter is incremented
- Timer reset/Kick (8bit addr, 21bit data) (active high is 21’h01 / low is 0)
    - CPU uses this to reset the timer counter
- Timeout Value (8bit addr, 21bit data)
    - Set the # of cycles to pass until automatic CPU reset is triggered
- CPU Reset (High/Low)
    - WDT output signal to trigger CPU reset

<u>Memory</u>

The simple internals of the memory module is seen in [Fig. 6]:

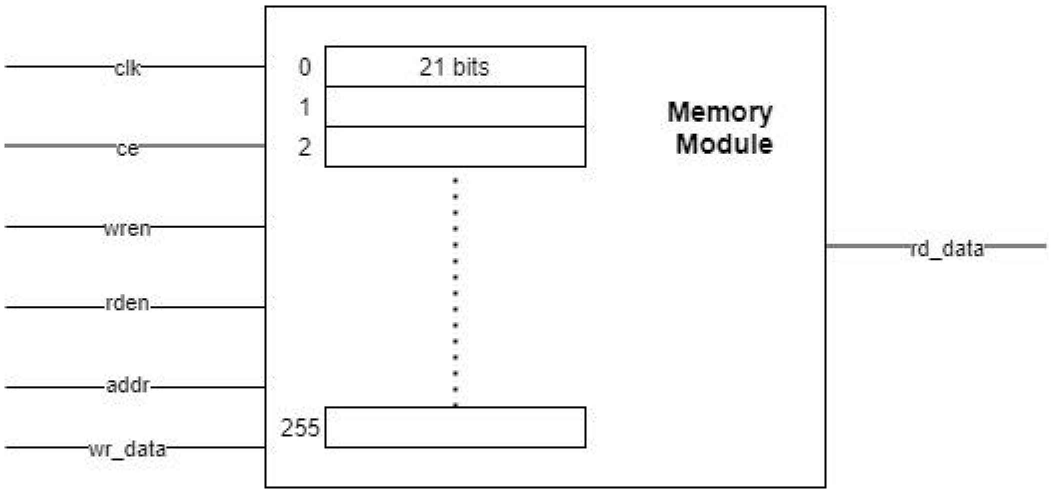


Fig. 6. Memory Module (Peripheral)

An explanation of the input and output signals of the module follows below:

- ce/wren/rden (1bit)
    - These signals determine if the cpu will be reading from, or writing to the memory. If reading from memory, the rd_data output signal will be updated with the requested data.
- addr (8bit)
    - This value will determine the register in memory to read from or write to.

- wr_data (21bit)
    - This value will hold the data to be written into memory.

C. Realistic Constraints

The constraints can be looked at from multiple angles. The first angle is what could be the reason for poor implementation of the watchdog timer in a SoC design:

1) It takes time and knowledge to design and test an effective watchdog. There are tips and tricks to implementing the watchdog, and the design may need to be slightly adapted to the particular system as knowing the timing of tasks could be important for some implementations to be aware of best times to insert a kick instruction or flag in the user code. An already late project could lead to a substandard implementation, or the cost for labor for such time and knowledge needed for a good implementation could simply be too expensive to be worthwhile.
2) One could also consider cost as a constraint. Could chips with superior hardware watchdog timers be more costly? If designing a new chip, could purchasing superior blackbox watchdog timer modules to fit in the design be costly? Cost is certainly a constraint for any project.

Another angle to consider is why have a watchdog timer module in the first place? Some engineers find a WDT to be unnecessary[WD4], preferring to focus on writing error free firmware instead.

3) I think a constraint on the belief that a WDT is unnecessary is the strong social, ethical, and health and safety concerns. Watchdog timers are used in embedded systems for industrial applications, and it isn’t a stretch to consider situations where the processor encounters an issue and the watchdog timeout is needed to put a system that controls dangerous hardware such as moving machinery, dangerous radiation, motors, etc. in a safe state. It may not be policy, but I think there is a certain expectation that the embedded system chosen would implement something as important as a WDT module as one method among many to prevent potential accidents to humans.
4) A major constraint in synthesizable RTL design is hardware constraints or manufacturability. For instance, FPGAs only have a certain amount of logic cells/units. This means that designs cannot be scalded indefinitely.
5) An ethical constraint that pertains to the CPU and WDT is the design's life span. Should hardware be designed to last for decades? While this might be great for consumers, it negatively affects designers, manufacturers and retailers economically. Should hardware designers emphasize profit over the design’s durability and longevity in the market? This question is definitely complicated and skewed by technology's rapid evolution. Sometimes new products quickly become obsolete because a new better technology is developed.

D. Engineering Standards

The use of the ARM AMBA 3 APB protocol has played a big part in the design. The communication from CPU to peripherals is primarily done thru the APB interface which has some trickle down effects:

- Master and Slave modules must be designed and tested to act as intermediaries between the CPU and peripherals. This master/slave designation dictates which device can initiate communication with another device. In this case, the design had the CPU as the master, and all other peripherals were designated as slaves. These master/slave modules also had to be designed to follow the APB specification: the state diagram was of significant importance.

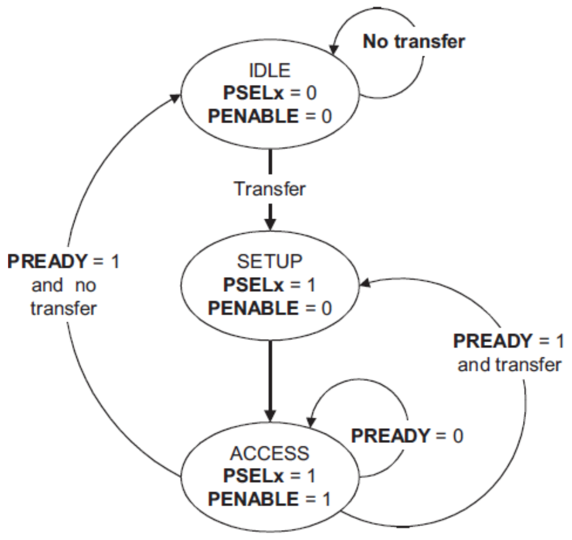


Fig. 7. APB Master Finite State Machine (FSM)

- It also played a factor in what peripherals were chosen: picking a common and appropriate APB peripheral was important such that in the time available a proper design could be done based on a multitude of available references.

The 1800-2017 IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language is also worth mentioning, as the project was implemented in this language.

E. Alternative Designs and Design Choices

There weren't several different designs made for the CPU and Watchdog Timer that underwent scrutiny; however, certain design choices were consciously made in order to simplify the project. In *section F* we describe some of these simplifications that introduced shortcomings in the design, whereas, in this section we will give some insight into options we could have considered to make the design more robust.

<u>CPU</u>

1) The CPU could have been implemented using a pipelined design. This would decrease the CPU's steady state cycles per instruction or CPI. However, pipelining the CPU would increase overall design complexity.
2) The ALU and the accumulator register were removed from the CPU design. This was due to the fact that none of the aforementioned instructions that the CPU can run require arithmetic operations. Removing the ALU decreased the overall complexity of the CPU.
3) The ISA has four different instruction formats. This mainly because the instructions are 32 bits wide. This means that for the instruction memory write instruction (opcode 111 or 3), the CPU will need to read two consecutive instructions. The first instruction alerts the CPU that this is an instruction memory write instruction. The second instruction is the actual instruction that needs to be written or moved in the instruction memory.

Three changes to the CPU design were described here. These changes could be made later to improve the CPU's usability and performance.

<u>Watchdog Timer</u>

1) A lock register could have been implemented that locks and unlocks write access to the other internal registers. This feature would provide some additional safety that the WDT registers would be overwritten during a system malfunction. This register could potentially use address x44 in the design, and expect different complicated data values to be written to activate and deactivate the locking mechanism.

2) The kick mechanism in the design expected a high value (21'h01) to be written to the register could have been replaced by a more complicated flag system that would allow catching the specific problem of an accidental jump out of the code area. The design could incorporate this feature by adding a few more registers that would directly correlate to flag registers. During the course of the user program, there would need to be instructions to set these flag registers to true as they are encountered. At some point there would be another special instruction that would cause those flag values to be checked- then if all the flag values were set to true, a successful "kick" would occur, otherwise a reset signal would need to be output, and the flags cleared for the next time.

3) Debug modes are popular for modules. To facilitate a debug mode, there could be an input signal that would stop the counter temporarily. This might be useful

when testing certain tasks and don't want to be concerned with the system timing to send kick signals when appropriate.

Three additions to the WDT design have been described, and while they each would add some benefit to the design, it should be apparent that the design choices that made it into the final design at least capture the main theme one would expect of a watchdog timer.

F. Technical analysis for system and subsystems

Due to time constraints the design was simplified. Below is a description of the simplifications made that would normally not appear in a production system:

Shortcomings of the CPU design:

- Not pipelined
    - Not pipelining the CPU increases the steady state CPI.
- No ALU
    - The CPU cannot execute arithmetic operations. This decreases the CPU's capabilities and usability.
- Only 7 instructions
    - The ISA does not utilize the 8th instruction slot. This slot could be used for another instruction, such as a memory peripheral move instruction.
- Instructions need to be loaded into the Instruction memory
    - This means that the design can only simulate an instruction memory write instruction. This is because the instruction to write must already be in the instruction memory. The CPU could have an additional stage called "READ", where the CPU waits for an instruction to come in from an input source and runs it.
- Complicated ISA
    - The ISA became more complicated because each instruction is only 32 bits. Increasing the instruction size to 64 bits would allow a single instruction to hold the new instruction that needs to be written.

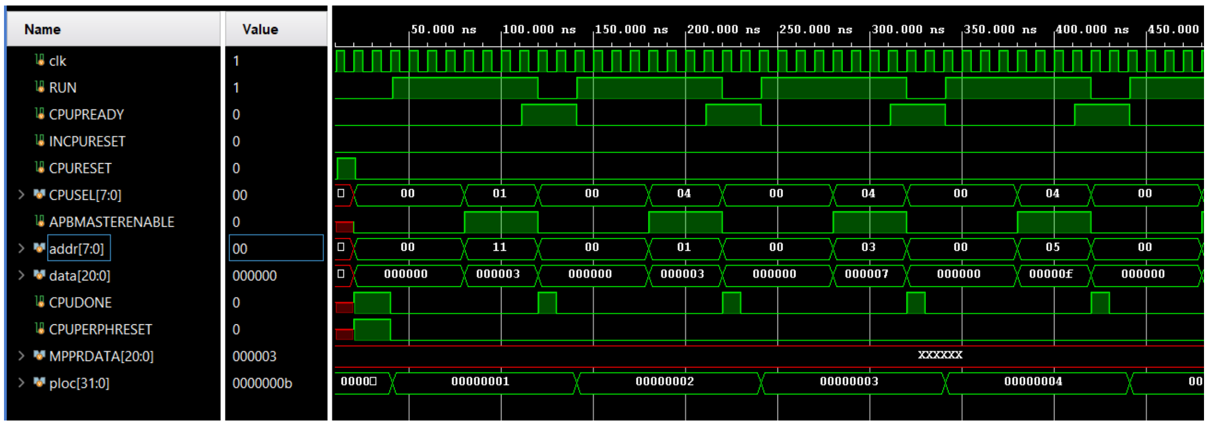
Shortcomings of the Watchdog Timer design:

- Simplified cpu reset/kick
    - A 0/1 signal could more easily be triggered accidentally thru cosmic bit flip, data corruption, etc
    - From a lack of using flags it won't catch an accidental jump out of code due to a bug/corruption

- Simplified follow-up action : only a cpu-reset is triggered
    - Triggering other peripherals (sounding an alarm, etc) isn't addressed
    - No reset tracking counter or debug data is stored

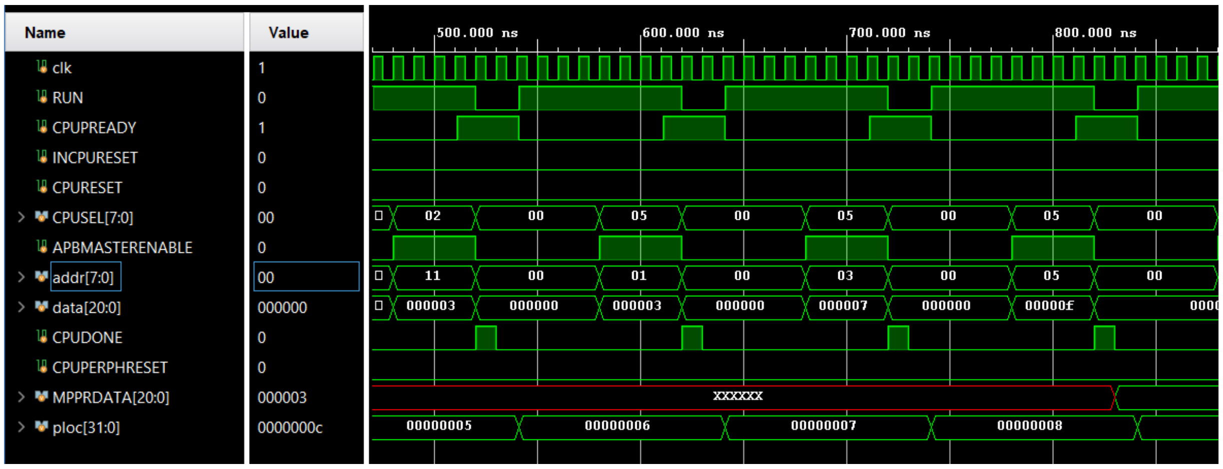
G. Design validation for system and subsystems

<u>**CPU and Memory UNIT testing**</u>



Name	Value
clk	1
RUN	1
CPUPREADY	0
INCPURESET	0
CPURESET	0
CPUSEL[7:0]	00
APBMASTERENABLE	0
addr[7:0]	00
data[20:0]	000000
CPUDONE	0
CPUPERPHRESET	0
MPPRDATA[20:0]	000003
ploc[31:0]	0000000b

Fig. 8. Write Instructions (opcode 001 and 100)



Name	Value
clk	1
RUN	0
CPUPREADY	1
INCPURESET	0
CPURESET	0
CPUSEL[7:0]	00
APBMASTERENABLE	0
addr[7:0]	00
data[20:0]	000000
CPUDONE	0
CPUPERPHRESET	0
MPPRDATA[20:0]	000003
ploc[31:0]	0000000c

Fig. 9. Read Instructions (opcode 010 and 101)

For both read and write instructions:

- APBMASTERENABLE is set to 1.
- CPUSEL is set to the corresponding instruction number/ opcode.
- addr and data push their corresponding values to master.
- CPUDONE becomes 1 once CPUREADY is 1.

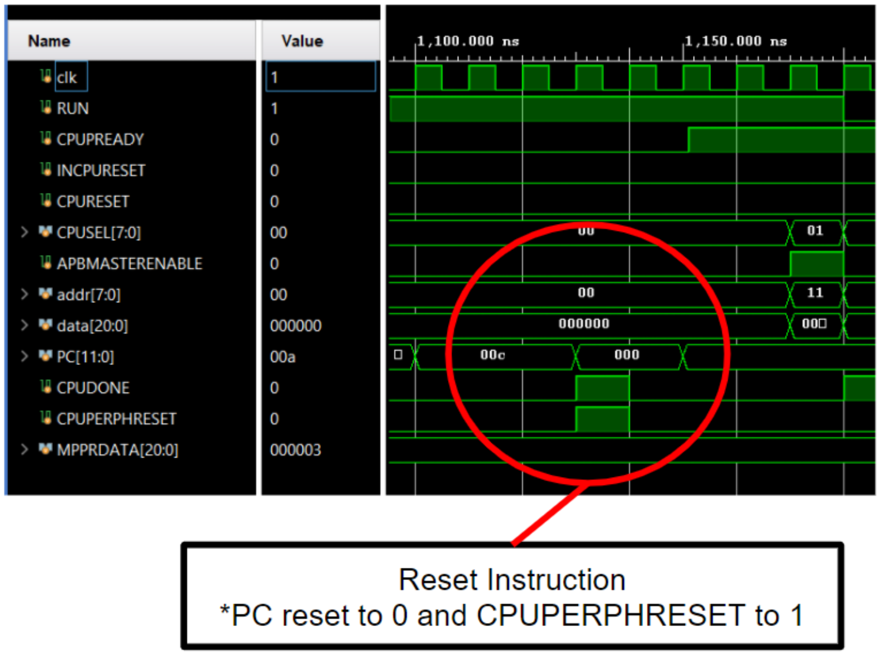


Fig. 10. Reset Instruction (opcode 000)

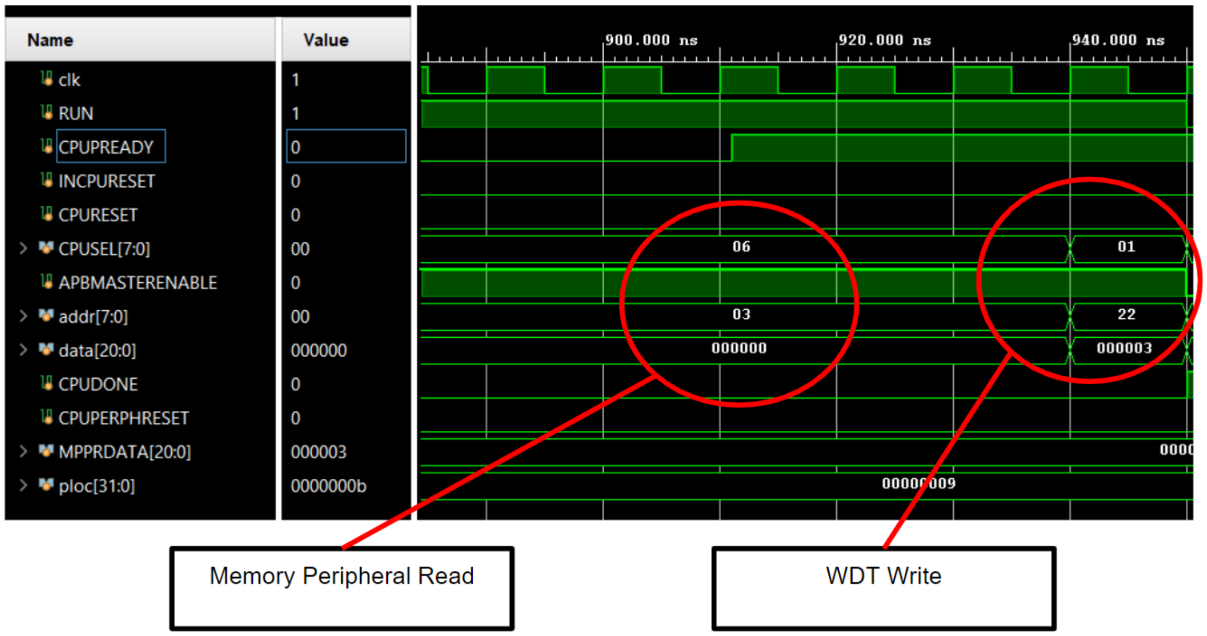
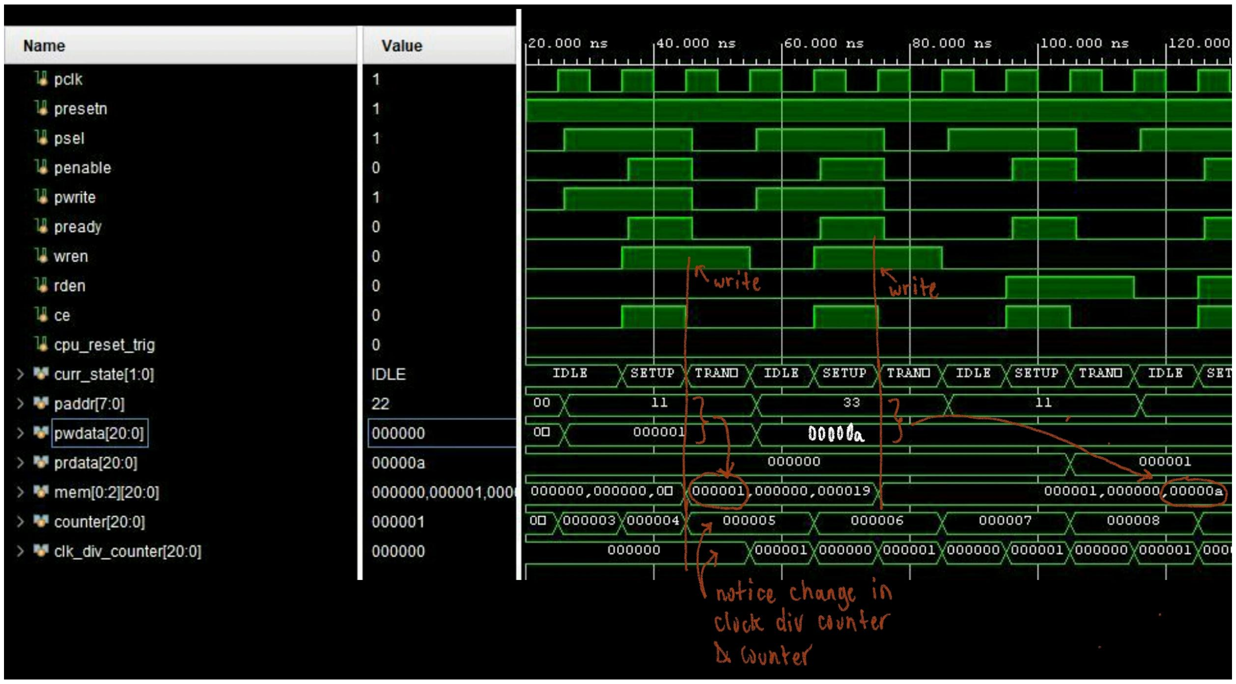


Fig. 11. Memory Peripheral Read to WDT Write (opcode 110)

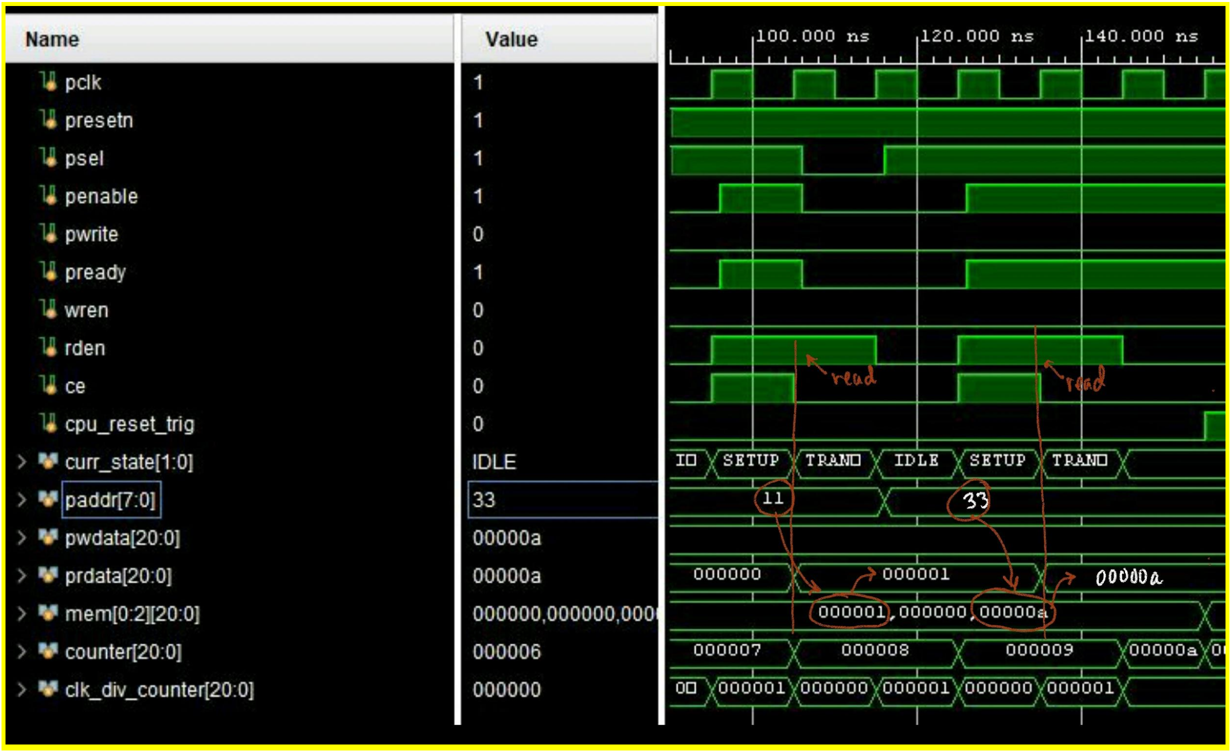
- The CPU reads 21 bit data from the memory peripheral and stores it in the MPPRDATA register.
- The value in MPPRDATA is then written to the appropriate WDT value.

Watchdog Timer UNIT testing



Name	Value
pclk	1
presetn	1
psel	1
penable	0
pwrite	1
pready	0
wren	0
rden	0
ce	0
cpu_reset_trig	0
curr_state[1:0]	IDLE
paddr[7:0]	22
pwdata[20:0]	000000
prdata[20:0]	00000a
mem[0:2][20:0]	000000,000001,000
counter[20:0]	000001
clk_div_counter[20:0]	000000

Fig. 12. Write to the clock division register then to the timeout register.



Name	Value
pclk	1
presetn	1
psel	1
penable	1
pwrite	0
pready	1
wren	0
rden	0
ce	0
cpu_reset_trig	0
curr_state[1:0]	IDLE
paddr[7:0]	33
pwdata[20:0]	00000a
prdata[20:0]	00000a
mem[0:2][20:0]	000000,000000,000
counter[20:0]	000006
clk_div_counter[20:0]	000000

Fig. 13. Reading back the clock division and timeout registers.

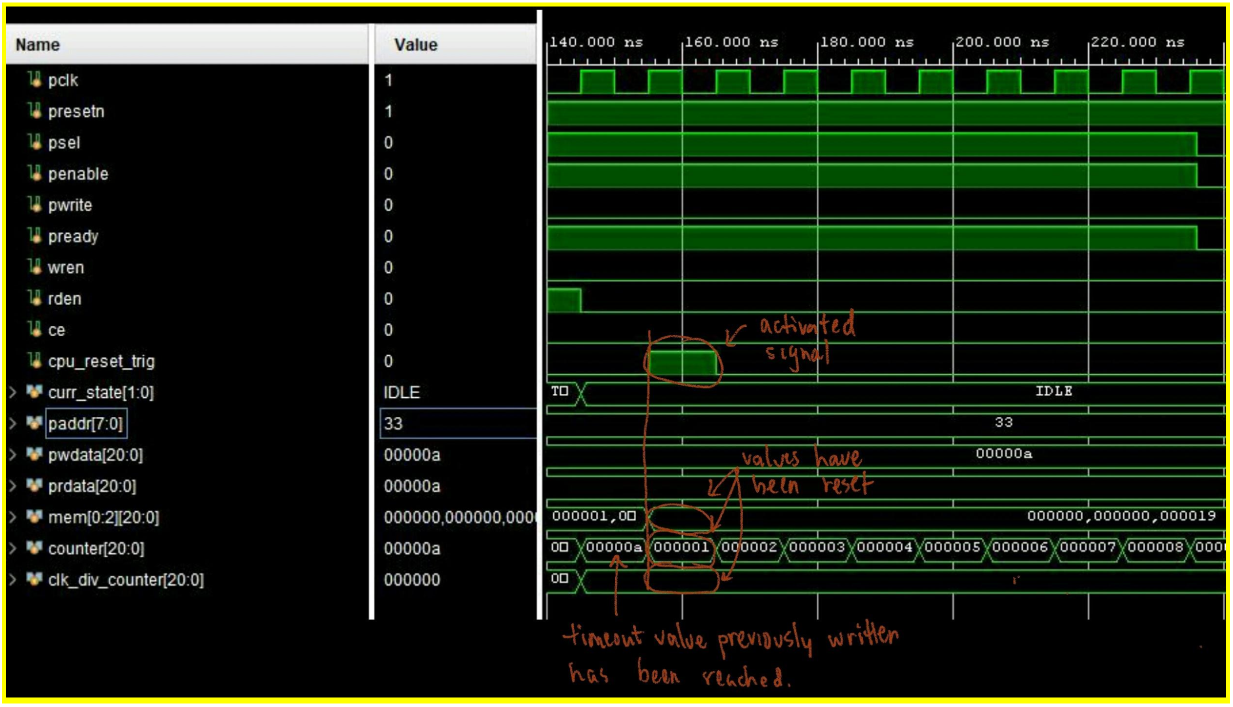


Fig. 14. Watching the counter timeout which then leads to a reset of the counter, clock division counter, the registers, and also activates the cpu_reset_trig signal to the CPU.

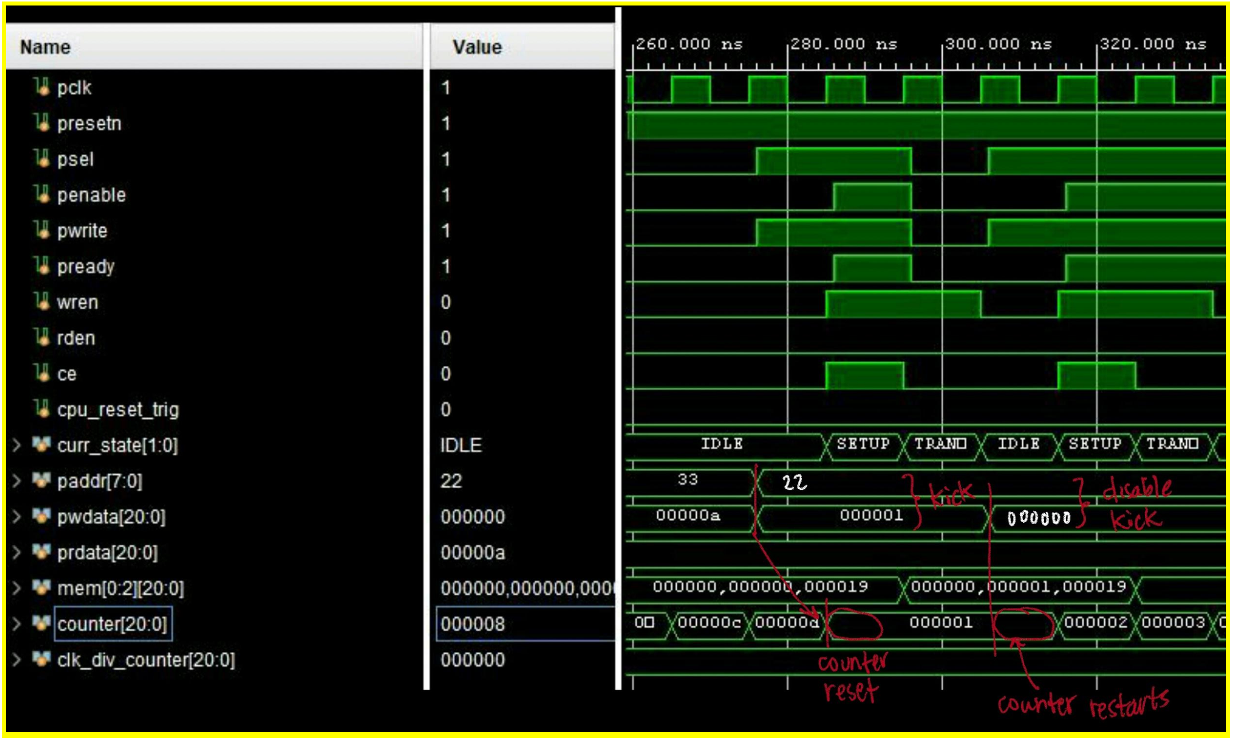
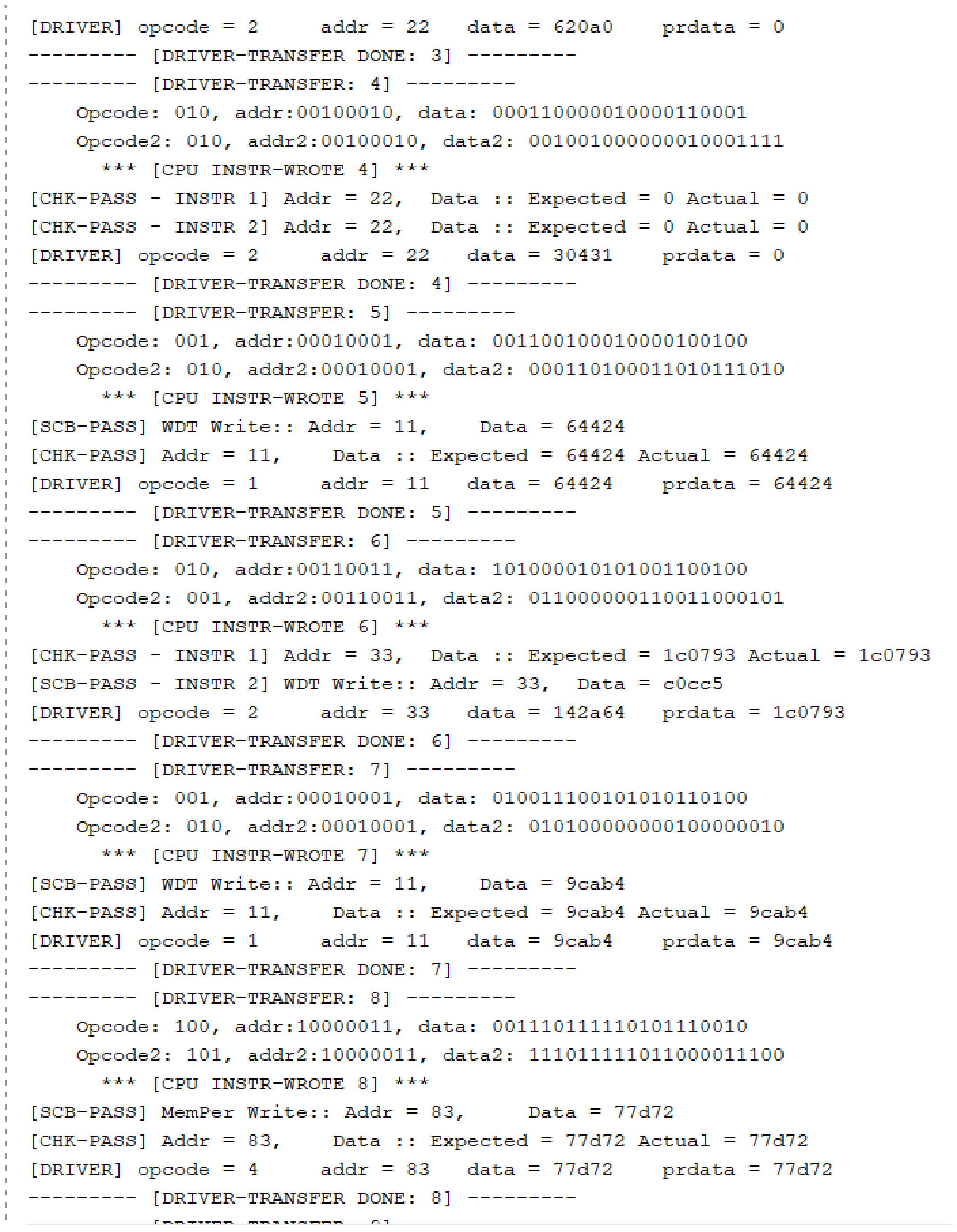


Fig. 15. Watching a CPU kick which resets the counter. Notice that the instruction somewhat goes through the APB so the states change, but the watchdog timer can act on a kick sooner than the transfer state.

Selected System Level Testing



```
[DRIVER] opcode = 2     addr = 22    data = 620a0     prdata = 0
--------- [DRIVER-TRANSFER DONE: 3] ---------
--------- [DRIVER-TRANSFER: 4] ---------
    Opcode: 010, addr:00100010, data: 00011000010000110001
    Opcode2: 010, addr2:00100010, data2: 00100100000010001111
      *** [CPU INSTR-WROTE 4] ***
[CHK-PASS - INSTR 1] Addr = 22,  Data :: Expected = 0 Actual = 0
[CHK-PASS - INSTR 2] Addr = 22,  Data :: Expected = 0 Actual = 0
[DRIVER] opcode = 2     addr = 22    data = 30431     prdata = 0
--------- [DRIVER-TRANSFER DONE: 4] ---------
--------- [DRIVER-TRANSFER: 5] ---------
    Opcode: 001, addr:00010001, data: 001100100010000100100
    Opcode2: 010, addr2:00010001, data2: 00011010011010111010
      *** [CPU INSTR-WROTE 5] ***
[SCB-PASS] WDT Write:: Addr = 11,     Data = 64424
[CHK-PASS] Addr = 11,     Data :: Expected = 64424 Actual = 64424
[DRIVER] opcode = 1     addr = 11    data = 64424     prdata = 64424
--------- [DRIVER-TRANSFER DONE: 5] ---------
--------- [DRIVER-TRANSFER: 6] ---------
    Opcode: 010, addr:00110011, data: 101000010101001100100
    Opcode2: 001, addr2:00110011, data2: 011000000110011000101
      *** [CPU INSTR-WROTE 6] ***
[CHK-PASS - INSTR 1] Addr = 33,  Data :: Expected = 1c0793 Actual = 1c0793
[SCB-PASS - INSTR 2] WDT Write:: Addr = 33,  Data = c0cc5
[DRIVER] opcode = 2     addr = 33    data = 142a64    prdata = 1c0793
--------- [DRIVER-TRANSFER DONE: 6] ---------
--------- [DRIVER-TRANSFER: 7] ---------
    Opcode: 001, addr:00010001, data: 010011100101010110100
    Opcode2: 010, addr2:00010001, data2: 010100000000100000010
      *** [CPU INSTR-WROTE 7] ***
[SCB-PASS] WDT Write:: Addr = 11,     Data = 9cab4
[CHK-PASS] Addr = 11,     Data :: Expected = 9cab4 Actual = 9cab4
[DRIVER] opcode = 1     addr = 11    data = 9cab4     prdata = 9cab4
--------- [DRIVER-TRANSFER DONE: 7] ---------
--------- [DRIVER-TRANSFER: 8] ---------
    Opcode: 100, addr:10000011, data: 00111011110101110010
    Opcode2: 101, addr2:10000011, data2: 11101111011000011100
      *** [CPU INSTR-WROTE 8] ***
[SCB-PASS] MemPer Write:: Addr = 83,     Data = 77d72
[CHK-PASS] Addr = 83,     Data :: Expected = 77d72 Actual = 77d72
[DRIVER] opcode = 4     addr = 83    data = 77d72     prdata = 77d72
--------- [DRIVER-TRANSFER DONE: 8] ---------
```

Fig. 16. This shows the messages generated by the driver/scoreboard/checker for the RANDOM test. Notice a mixture of watchdog timer instructions and memory module instructions. For the random tests the watchdog read is set to follow a watchdog write for convenience.

H. Test Plan

The previous section G holds all the screenshots for system validation, but in this section we explain our test strategy that resulted in those screenshots. The testing is broken down into unit level, and system level testing, with similar tests for each to verify module functionality.

Unit Level

- CPU - 2 tests
  - cpu reset, cpu instruction memory write
- Watchdog Timer - 4 basic tests
  - read/write registers, successful timeout, successful kick, successful reset (from cpu)
- Memory Module - 1 basic test
  - read/write registers

System Level

- Random Tests
- Directed Tests - 2 detailed tests
  - Multiple writes to WDT, read back the values, reset instruction, read back WDT (default) values. Kick WDT to reset timer countdown. Let WDT timeout.
  - Write data to memory module, use data in memory module to write to WDT registers. Read back WDT register values.

[Fig. 17] describes how the system was coded:

- TEST is a System Verilog *program*
- ENVIRONMENT, TRANSACTION, GENERATOR, DRIVER, MONITOR, and SCOREBOARD/CHECKER are all System Verilog *classes*
- MEM_INF, CPU_IF, SYS_IF are System Verilog *interfaces*
- Most of the classes were inserted into a System Verilog *package* for ease in compilation in Xilinx Vivado
- All other files were System Verilog *modules*

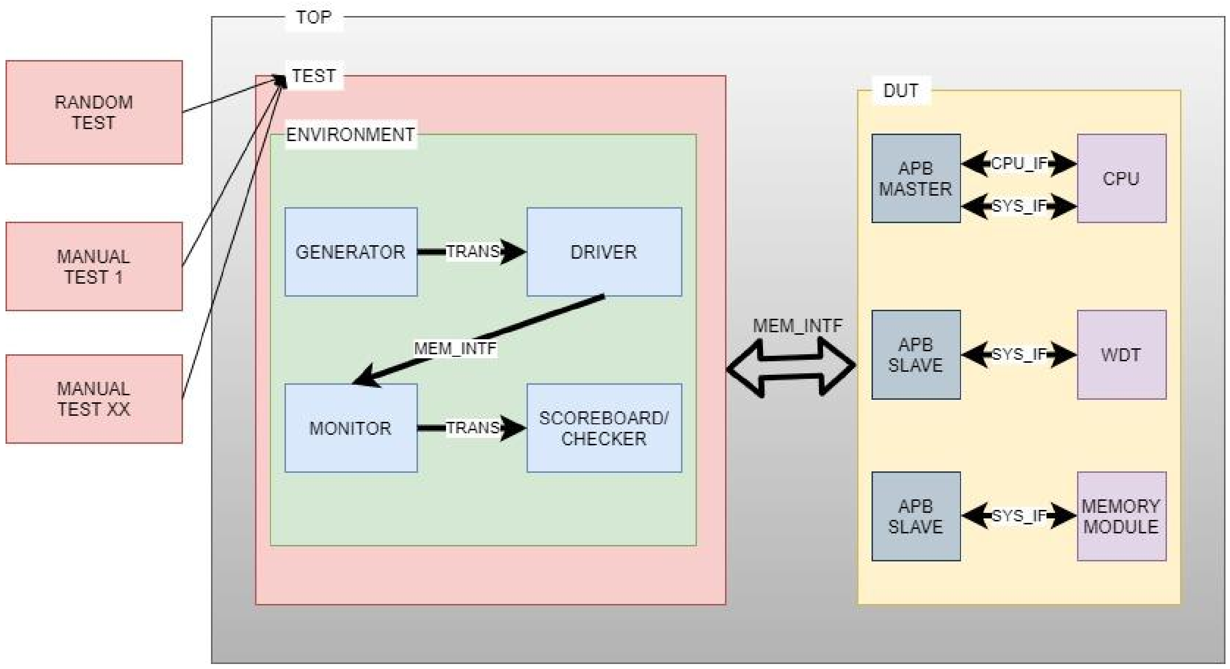


Fig. 17. High level diagram of the system level testing environment [8]

The testing consists of manipulating two instructions at a time with the following six components:

opcode / addr / data
opcode_2 / addr_2 / data_2

Once the initial instructions are filled into the CPU ram the system can “start”.

I. Project Planning and Management

Our project management plan involves the following to stay on track:

1. A GroupMe room for asynchronous messaging which is more convenient than simply emailing.
2. Private zoom meetings during the week to keep in touch.
3. Using Google Doc/Sheets/Slides for sharing and multi-user editing of our documents.

The task breakdown is centered around splitting up the Processor/APB Master related information to Richard and the Watchdog Timer/APB Slave information to Noel.

The Gantt chart schedule was adhered to for phase one of the project (up to the midterm presentation); however, the technical issues in phase 2 involving the integration of the

CPU and watchdog timer designs into the system environment took time and became the main focus. Auxiliary tasks were then continuously pushed back.

V. Conclusions

The approach taken for the Watchdog timer module was a simple single-stage design with a very simple high/low signal for resetting the timer. This simplicity would make for a poor implementation for an embedded device due to lacking persistent debug data stored in the WDT module, and the timer countdown reset mechanism is too easily triggered allowing a process going haywire to potentially keep resetting the WDT accidentally. That said, the simple model is still helpful in exploring the APB specification and interaction between APB Masters and APB Slaves by using the Processor, Memory module and WDT peripheral.

The CPU is a simple processor, which follows the fetch-decode-execute instruction cycle. The processor uses a 32 bit instruction set with a 3 bit opcode. Thus, this system can execute up to eight different instructions. If more instructions are to be added, then more bits would need to be allocated for the opcode. The CPU drives the address and data, which are contained in the 32 bit instructions, to the APB master. The CPU is also connected to the WDT peripheral. When the WDT's timeout counter finishes, the WDT drives an interrupt/reset signal to the CPU. This resets the CPU and the rest of the system.

One challenge involved satisfying requirements for the Processor/Peripheral/APB Spec while simplifying things to a manageable level. This was solved by making use of existing designs and just paring them down. Once we did that other concerns cropped up: for example when simplifying the ARM design, most of the signals going through the APB bus were eliminated, while only the independent signals remained that went through another path between Processor and Peripheral. The solution was to move some signals to be transferred through the APB bus as the APB bus is an integral part of the project.

Another challenge was realizing that even though tasks can be broken down and assigned to different individuals, when tasks intersect we need to come together and hash out the interactions required- like between the processor and peripheral for example. Doing this while trying to meet other task deadlines and work independently takes some planning and scheduling, but was solved with regularly communicating through GroupMe or Zoom as needed.

Having a good management plan is a big lesson learned. With a big project with many parts, it was helpful and also a relief to have small manageable tasks and deadlines to complete.

VI. References

[1] “Von Neumann Architecture,” *Computer Science GCSE GURU*. [Online]. Available: https://www.computerscience.gcse.guru/theory/von-neumann-architecture. [Accessed: 11-Dec-2020].

[2] “AMBA APB Protocol Specification Version 2.0,” *Documentation – Arm Developer*. [Online]. Available: https://developer.arm.com/documentation/ddi0270/b/. [Accessed: 11-Dec-2020].

[3] “IEEE 1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language,” *IEEE SA - The IEEE Standards Association - Home*. [Online]. Available: https://standards.ieee.org/standard/1800-2017.html. [Accessed: 11-Dec-2020].

[4] “ARM Watchdog Module (SP805) Technical Reference Manual,” *Documentation – Arm Developer*. [Online]. Available: https://developer.arm.com/documentation/ddi0270/b/. [Accessed: 11-Dec-2020].

[5] “Laboratory Exercise 9 A Simple Processor.” Intel Corporation, 2016.

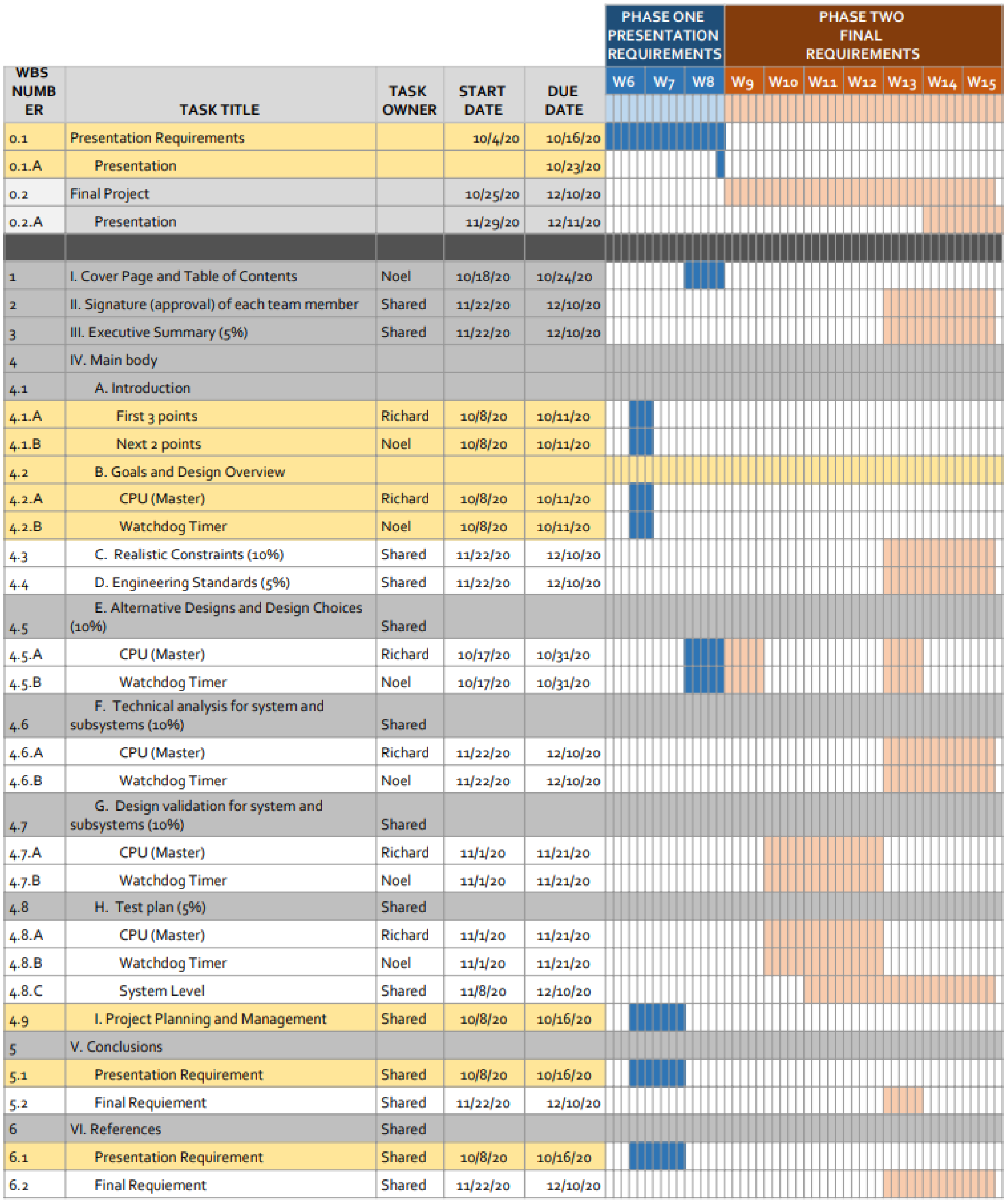
[6] J. Ganssle, “Great Watchdog Timers For Embedded Systems,” *Designing Great Watchdog Timers for Embedded Systems*. [Online]. Available: http://www.ganssle.com/watchdogs.htm. [Accessed: 11-Dec-2020].

[7] “Watchdog timer,” *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/Watchdog_timer. [Accessed: 11-Dec-2020].

[8] “SystemVerilog TestBench Example 01,” *Verification Guide*. [Online]. Available: https://verificationguide.com/systemverilog-examples/systemverilog-testbench-example-01/. [Accessed: 11-Dec-2020].

Appendices

A. GANTT chart



WBS NUMBER	TASK TITLE	TASK OWNER	START DATE	DUE DATE	PHASE ONE PRESENTATION REQUIREMENTS W6	W7	W8	PHASE TWO FINAL REQUIREMENTS W9	W10	W11	W12	W13	W14	W15
0.1	Presentation Requirements		10/4/20	10/16/20	🟦	🟦	🟦							
0.1.A	Presentation			10/23/20			🟦							
0.2	Final Project		10/25/20	12/10/20				🟧	🟧	🟧	🟧	🟧	🟧	🟧
0.2.A	Presentation		11/29/20	12/11/20									🟧	🟧
1	I. Cover Page and Table of Contents	Noel	10/18/20	10/24/20			🟦							
2	II. Signature (approval) of each team member	Shared	11/22/20	12/10/20								🟧	🟧	🟧
3	III. Executive Summary (5%)	Shared	11/22/20	12/10/20								🟧	🟧	🟧
4	IV. Main body													
4.1	A. Introduction													
4.1.A	First 3 points	Richard	10/8/20	10/11/20	🟦	🟦								
4.1.B	Next 2 points	Noel	10/8/20	10/11/20	🟦	🟦								
4.2	B. Goals and Design Overview													
4.2.A	CPU (Master)	Richard	10/8/20	10/11/20	🟦	🟦								
4.2.B	Watchdog Timer	Noel	10/8/20	10/11/20	🟦	🟦								
4.3	C. Realistic Constraints (10%)	Shared	11/22/20	12/10/20								🟧	🟧	🟧
4.4	D. Engineering Standards (5%)	Shared	11/22/20	12/10/20								🟧	🟧	🟧
4.5	E. Alternative Designs and Design Choices (10%)	Shared												
4.5.A	CPU (Master)	Richard	10/17/20	10/31/20			🟦	🟧				🟧		
4.5.B	Watchdog Timer	Noel	10/17/20	10/31/20			🟦	🟧				🟧		
4.6	F. Technical analysis for system and subsystems (10%)	Shared												
4.6.A	CPU (Master)	Richard	11/22/20	12/10/20								🟧	🟧	🟧
4.6.B	Watchdog Timer	Noel	11/22/20	12/10/20								🟧	🟧	🟧
4.7	G. Design validation for system and subsystems (10%)	Shared												
4.7.A	CPU (Master)	Richard	11/1/20	11/21/20					🟧	🟧	🟧			
4.7.B	Watchdog Timer	Noel	11/1/20	11/21/20					🟧	🟧	🟧			
4.8	H. Test plan (5%)	Shared												
4.8.A	CPU (Master)	Richard	11/1/20	11/21/20					🟧	🟧	🟧			
4.8.B	Watchdog Timer	Noel	11/1/20	11/21/20					🟧	🟧	🟧			
4.8.C	System Level	Shared	11/8/20	12/10/20						🟧	🟧	🟧	🟧	🟧
4.9	I. Project Planning and Management	Shared	10/8/20	10/16/20	🟦	🟦								
5	V. Conclusions													
5.1	Presentation Requirement	Shared	10/8/20	10/16/20	🟦	🟦								
5.2	Final Requiement	Shared	11/22/20	12/10/20								🟧		
6	VI. References	Shared												
6.1	Presentation Requirement	Shared	10/8/20	10/16/20	🟦	🟦								
6.2	Final Requiement	Shared	11/22/20	12/10/20								🟧	🟧	🟧

B. Watchdog UNIT test code

1. Watchdog Timer Module

```
// NOTE: all active signals are HIGH (1)
module WDT (pclk, paddr, pwdata, wren, rden, prdata, cpu_forced_reset, cpu_reset_trig);

input pclk, wren, rden, cpu_forced_reset;
input [7:0] paddr;
input [20:0] pwdata;
output reg [20:0] prdata;
output reg cpu_reset_trig;

reg [20:0] mem [0:2];
reg [7:0] new_addr;
reg [20:0] counter, clk_div_counter;
bit default_counter;

// parameters ////////////////////////////////////////////////////////////////////////////////
// cpu instructions use these addresses to set internal registers in the WDT
parameter CLOCK_DIVIDER_ADDR = 8'h11;
parameter KICK_ADDR = 8'h22;
parameter TIMEOUT_ADDR = 8'h33;

// internal register memory has a much lower, contiguous set of addresses
parameter INT_REG_CLK_DIV = 0;
parameter INT_REG_KICK = 1;
parameter INT_REG_TIMEOUT = 2;

// default values for the internal registers upon generic reset
parameter DEFAULT_CLK_DIV = 21'h0; // no clock division
parameter DEFAULT_TIMEOUT = 21'h19; // 25 cycles default
parameter DEFAULT_KICK = 21'h0;

parameter EXPECTED_KICK_VALUE = 21'h01; // expected active signal for a "cpu  kick"

initial mem[INT_REG_CLK_DIV] = DEFAULT_CLK_DIV;
initial mem[INT_REG_TIMEOUT] = DEFAULT_TIMEOUT;
initial mem[INT_REG_KICK] = DEFAULT_KICK;
initial counter = 0;
initial clk_div_counter = 0;

// assignment statements //////////////////////////////////////////////////////////////////////
//assign rd_data = mem[new_addr]; // rd_data always populated. Ignore  in-between values

// always blocks /////////////////////////////////////////////////////////////////////////////////
// adjust input address from CPU (addr) to match the internal register memory addresses
always @ (posedge pclk) begin
if (paddr == CLOCK_DIVIDER_ADDR)
new_addr <= INT_REG_CLK_DIV;
```

```
else if (paddr == KICK_ADDR)
new_addr <= INT_REG_KICK;
else if (paddr == TIMEOUT_ADDR)
new_addr <= INT_REG_TIMEOUT;
else
new_addr <= 5; // invalid address for the watchdog

end // always block

// logic for reading or writing writing to internal (register) memory or rd_data output
always @ (posedge pclk iff new_addr != 5 ) begin
if (cpu_forced_reset == 1 || counter == mem[INT_REG_TIMEOUT]) begin // || default_counter != 1) begin // Set defaults (stored value not important for the  kick (use immediately))
mem[INT_REG_CLK_DIV] <= DEFAULT_CLK_DIV;
mem[INT_REG_TIMEOUT] <= DEFAULT_TIMEOUT;
mem[INT_REG_KICK] <= DEFAULT_KICK;
default_counter <= 1; // used to initialize registers/variables to default values at first clock cycle
end
else if (wren)
mem[new_addr] <= pwdata; // write the value from the cpu instruction into internal (register) memory
else if (rden)
prdata <= mem[new_addr];
end // always block

// counter logic
always @ (posedge pclk) begin
if (cpu_forced_reset == 1) // || default_counter != 1)
counter <= 1;
else if ((paddr == KICK_ADDR && pwdata == EXPECTED_KICK_VALUE) || counter == mem[ INT_REG_TIMEOUT]) // check for a "kick" from cpu or timeout
counter <= 1;
else if (clk_div_counter == mem[INT_REG_CLK_DIV])
counter <= counter + 1; // increment the counter
end // always block

// clk div logic
always @ (posedge pclk) begin
if (cpu_forced_reset == 1 || clk_div_counter == mem[INT_REG_CLK_DIV]) // ||  default_counter != 1)
clk_div_counter <= 0; // roll clock division  counter back to 0
else if ((paddr == KICK_ADDR && pwdata == EXPECTED_KICK_VALUE) || counter == mem[ INT_REG_TIMEOUT]) // check for a "kick" from cpu or timeout
clk_div_counter <= 0; // roll clock division  counter back to 0
else
clk_div_counter <= clk_div_counter + 1;
end // always block

// cpu reset trigger logic
always @ (posedge pclk) begin
if (cpu_forced_reset == 1 || cpu_reset_trig == 1)
cpu_reset_trig <= 0;
else if (counter == mem[INT_REG_TIMEOUT]) // check if cpu reset trigger
```

```
        should be asserted
cpu_reset_trig <= 1;
end

endmodule : WDT
```

2. APB Module

```
`timescale 1ns / 1ps

module APB_Slave_WDT(pclk, presetn, paddr, psel, penable, pwrite,
  pwdata, wait_time, cpu_forced_reset, prdata, pready, wren, rden,
  prdata_out, ce);

input logic pclk, presetn, psel, penable, pwrite,
cpu_forced_reset; 6 input logic [7:0] paddr;
input logic [20:0] pwdata;
input logic [3:0] wait_time;
input logic [20:0] prdata;
output logic [20:0] prdata_out;
output logic pready, wren, rden;
output logic ce;

logic [3:0] wait_counter;

// enums
  ///////////////////////////////////////////////////////////////////////////
  /////////////// //////////
enum logic [1:0]
{
IDLE = 2'b00,
SETUP = 2'b01,
WAITT = 2'b10,
TRANSFER = 2'b11
} next_state, curr_state;


// assign statements
  ///////////////////////////////////////////////////////////////////////////
/////////////// 27 assign pready = psel && penable && (wait_counter ==
wait_time);
assign prdata_out = pready && prdata;
// always blocks
  ///////////////////////////////////////////////////////////////////////////
  /////////////// /
// next state handling
always_comb begin
case (curr_state)
IDLE : begin
ce = 0; wren = 0; rden = 0;
if ((psel) && (~penable))
next_state = SETUP; //2'b01;
else
next_state = IDLE; //2'b00;
end
```

```
SETUP : begin
wren = pwrite; rden = ~pwrite;
if ((psel) && (penable) && (~pready) && (wait_counter < wait_time)) begin
next_state = WAITT; //2'b10;
ce = 0;
end
else begin
next_state = TRANSFER; //2'b11;
ce = 1;
end
end
WAITT : begin
if ((psel) && (penable) && (wait_counter === wait_time)) begin //(pready)) begin
next_state = TRANSFER; //2'b11;
ce = 1;
end
else
next_state = WAITT; //2'b10;
end
TRANSFER : begin
ce = 0;
if ((psel) && (~penable) && (pready))
next_state = SETUP; //2'b01;
else
next_state = IDLE; //2'b00;
end
endcase
end // always_comb

// wait counter handling
always_ff @ (posedge pclk or presetn) begin
if (~presetn)
wait_counter <= 0;
else if (next_state == SETUP)
wait_counter <= 0;
else if ((next_state == WAITT) && (wait_counter < wait_time))
++wait_counter;
end // always_ff

// logic for current state status
always_ff @ (posedge pclk) begin
if (~presetn)
curr_state <= IDLE;
else
curr_state <= next_state;
end // always_ff

endmodule : APB_Slave_WDT
```

3. Testbench

```
`timescale 1ns / 1ps

```

```
// the unit test for the WDT and APB slave module
module apb_slave_wdt_testbench;

bit pclk, presetn, psel, penable, pwrite, pready, wren, rden,
  cpu_forced_reset, cpu_reset_trig, ce;
bit [7:0] paddr;
bit [20:0] pwdata, prdata, prdata_out;
bit [3:0] wait_time;


// DUT instantiation
APB_Slave_WDT apb_wdt_inst1 (.*);
WDT wdt_inst1(.*);

// setup clock
always #5 pclk = ~pclk;

initial begin
pclk = 0;
wait_time = 0; // tested 0 case and 3

// Drive reset
presetn <= 0;
repeat(2) @(posedge pclk)
#1;
presetn <= 1; // disable reset

// setup for first write clock divider
@(posedge pclk)
#1;
paddr <= 8'h11;
pwdata <= 21'h01;
pwrite <= 1'b1;
psel <= 1'b1;

// Toggle PEnable and thus PReady
@(posedge pclk)
#1;
penable <= 1'b1;
repeat (wait_time) @(posedge pclk);
// write transaction should occur

// transition
@(posedge pclk)
#1;
penable <= 1'b0;
pwrite <= 1'b0;
psel <= 1'b0;

// setup for second write to timeout value
@(posedge pclk)
#1;
paddr <= 8'h33;
pwdata <= 21'h0A; // 10 cc
pwrite <= 1'b1;
psel <= 1'b1;

```

```verilog
// Toggle PEnable and thus PReady
@(posedge pclk)
#1;
penable <= 1'b1;
repeat (wait_time) @(posedge pclk);
// write transaction should occur

// transition
@(posedge pclk)
#1;
penable <= 1'b0;
pwrite <= 1'b0;
psel <= 1'b0;

// read what is in clock divider reg
@(posedge pclk)
#1;
paddr <= 8'h11;
psel <= 1'b1;

// Toggle PEnable and PReady
@(posedge pclk)
#1;
penable <= 1'b1;
// read transaction should occur

// transition
@(posedge pclk)
#1;
penable <= 1'b0;
pwrite <= 1'b0;
psel <= 1'b0;

// read what is in timeout reg
@(posedge pclk)
#1;
paddr <= 8'h33;
psel <= 1'b1;

// Toggle PEnable and PReady
@(posedge pclk)
#1;
penable <= 1'b1;
// read transaction should occur

// expect a cpu_reset_trig after few cycles
repeat (10)@(posedge pclk);
#1;

// transition
@(posedge pclk)
#1;
penable <= 1'b0;
pwrite <= 1'b0;
psel <= 1'b0;

// wait few cycles then write to timer kick register (direct from CPU) and
```

```
  watch  internal wdt counter reset
repeat (4)@(posedge pclk);
#1;
pwrite <= 1'b1;
psel <= 1'b1;
paddr <= 8'h22;
pwdata <= 21'h01;

// Toggle PEnable and thus PReady
@(posedge pclk)
#1;
penable <= 1'b1;
repeat (wait_time) @(posedge pclk);
// write transaction should occur

// transition
@(posedge pclk)
#1;
penable <= 1'b0;
pwrite <= 1'b0;
psel <= 1'b0;

// disable kick
@(posedge pclk)
#1;
paddr <= 8'h22;
pwdata <= 21'h00;
pwrite <= 1'b1;
psel <= 1'b1;

// Toggle PEnable and thus PReady
@(posedge pclk)
#1;
penable <= 1'b1;
// write should occur

end

endmodule : apb_slave_wdt_testbench
```

C. CPU code

1. Cpu Module

```
module cpu(input logic RUN, input logic CPUREADY,
input logic CCLK,input logic INCPURESET, output logic APBMASTERENABLE,
output logic [7:0] addr, output logic [20:0]
data, output logic[7:0] CPUSEL,
input logic CPURESET, output logic CPUDONE, output logic CPUPERPHRESET,
input logic [20:0] MPPRDATA);

//Define constants
`define data_size 32
`define addr_size 12
`define decoder_output_size 8
//Write, Read from main memeory
logic ce, mem_wren, mem_rden, mem_cpu_ready, MWRITE;
logic [`data_size - 1:0] wr_instruction, rd_instruction;

//Program counter, Memory access register
logic [`addr_size - 1:0] PC, MAR;
//Memory data register, Instruction register
logic [`data_size - 1:0] MDR, IR;

//Store value returned by Mem Peripheral
logic[`data_size - 1:0] MPERR;

//Decode the instruction
logic instr_decoder_enable;
logic [`decoder_output_size - 1:0] instr_decoder_output;

// Store the current and next state
enum logic [4:0]{
FETCH1 = 5'b000,
FETCH2 = 5'b001,
DECODE = 5'b010,
EXECUTE = 5'b011,
IDLE = 5'b100,//Final or rest state
MEMORY1 = 5'b101,// Special State for writing to the Instruction mem or wdt with  data from mem peripheral
MEMORY2 = 5'b110, // Special State for writing to the Instruction mem
WRITEBACK1 = 5'b111, // Special State for writing to the Instruction mem
WRITEBACK2 = 5'b1000, // Special State for writing to the Instruction mem
RESET = 5'b1001 // Special State for reseting the cpu and perph
} next_state, curr_state;

task cpu_reset;
begin
CPUPERPHRESET <= 1;
CPUDONE <= 1;
PC <= 0;
MAR <= 0;
APBMASTERENABLE <= 0;
addr <= 0;
data <= 0;
CPUSEL <= 0;
MWRITE <= 0;
end
endtask: cpu_reset;

task new_instr;
begin
MAR <= PC;
```

```
APBMASTERENABLE <= 0;
addr <= 0;
data <= 0;
CPUSEL <= 0;
MWRITE <= 0;
CPUDONE <= 1;
end
endtask:new_instr

task readyToRW(input read_on, input write_on);//Push values for reading and writting to  Instruction mem
begin
mem_wren <= write_on;
mem_rden <= read_on;
ce <= 1;
end
endtask:readyToRW

//Instruction memory
mainMem imemory(MAR, ce, mem_wren, mem_rden, wr_instruction, rd_instruction, CCLK, mem_cpu_ready);
//Instruction Decoder
threeBitDecoder instrDecoder(IR[31:29],instr_decoder_enable, instr_decoder_output); //8-bit out

//Current State logic
always_ff@(posedge CCLK or posedge CPURESET or posedge INCPURESET) begin
if(CPURESET)begin //CPU Reset signal
curr_state <= RESET;
end else if(INCPURESET) begin //Interrupt signal from peripheral
curr_state <= RESET;
end else begin
curr_state <= next_state;
end
end

//Drive signals
always_ff@(posedge CCLK) begin
case(curr_state)
FETCH1:begin//Fetch the next instruction from the Instruction memory
CPUDONE <= 0;
CPUPERPHRESET <= 0;
if(RUN)begin
MAR <= PC;
end else begin
MAR <= MAR;
end
end
FETCH2:begin//Fetch the next instruction from the Instruction memory
PC <= PC + 1;
MDR <= rd_instruction;
end

DECODE:begin
MAR <= PC;
IR <= MDR;
```

```
end

EXECUTE:begin
if(!IR) begin // No instruction
APBMASTERENABLE <= 0;
PC <= PC;
end else if (IR) begin
  if (instr_decoder_output == 8'h01 || instr_decoder_output == 8'h02 ||
  instr_decoder_output == 8'h04 || instr_decoder_output == 8'h05) begin
CPUSEL <= instr_decoder_output;
addr <= IR[28:21];
data <= IR[20:0];
APBMASTERENABLE <= 1;
PC <= PC;
   end else if(instr_decoder_output == 8'h03) begin//Instruction Mem
                                    Write Instruction
PC <= PC + 1;
APBMASTERENABLE <= 0;
   end else if (instr_decoder_output == 8'h06) begin
  CPUSEL <= instr_decoder_output; //Read from Mem  Peripheral
addr <= IR[7:0];
data <= 0; //Dont care
APBMASTERENABLE <= 1;
PC <= PC;
end else if(instr_decoder_output == 8'h07) begin //Invalid instruction
PC <= PC;
APBMASTERENABLE <= 0;
end else if(!instr_decoder_output) begin //Reset
end
end else begin //Unknown State
end
end

//Stages for special write instructions i.e.
OpCode=011, 110
MEMORY1:begin
//Read from mem peripheral and write to wdt
if(instr_decoder_output == 8'h01) begin
IR[20:0] <= MPERR;
end else
IR <=IR;
end
MEMORY2:begin
MDR <= rd_instruction;
end
WRITEBACK1:begin
wr_instruction <= MDR;
MAR[11:8] <= 4'b0000;
MAR[7:0] <= IR[28:21];
MWRITE <= 1;
end
WRITEBACK2:begin
end

//State for waiting for current instruction to finish
IDLE:begin
MWRITE <= 0;
```

```
if(!CPUPREADY && !mem_cpu_ready) begin
end else begin
if(instr_decoder_output == 8'h06) begin
MPERR <= MPPRDATA; //Data Read from Mem peripheral
IR[31:29] <= 3'b001; //WDT Instruction
end else if (instr_decoder_output != 8'h06) begin
new_instr;
end
end
end

//State to handle reset instructions
RESET:begin
cpu_reset;
end
endcase
end

//Next state logic/ Control Unit FSM
always_comb begin
case(curr_state)
FETCH1:begin//Fetch the next instruction from the Instruction memory
if(RUN)begin
readyToRW(!MWRITE, MWRITE);//Prepare memory to read next instruction
next_state = curr_state.next;
end else begin
next_state = curr_state;
end
end
FETCH2:begin//Fetch the next instruction from the Instruction memory
next_state = curr_state.next;
end

DECODE:begin
instr_decoder_enable <= 1;
readyToRW(!MWRITE, MWRITE);//Prepare memory to read next instruction
next_state = curr_state.next;
end

EXECUTE:begin
if(!IR) begin // No instruction
next_state = IDLE;//IDLE, instruction is done
end else if(IR) begin
if(instr_decoder_output == 8'h01 || instr_decoder_output == 8'h02 ||
instr_decoder_output == 8'h04 || instr_decoder_output == 8'h05) begin
next_state = IDLE;//IDLE, instruction is done
end else if(instr_decoder_output == 8'h03) begin//Instruction Mem  Write Instruction
next_state = MEMORY1;//Memory Write Instruction requires  additional stages
end else if (instr_decoder_output == 8'h06) begin
next_state = IDLE;
end else if(instr_decoder_output == 8'h07) begin //Invalid instruction
next_state = curr_state;
end else if(!instr_decoder_output) begin //Reset
next_state = RESET;//RESET, instruction is done
end
end else begin //Unknown State
```

```
next_state = curr_state.first;
end
end

//Stages for special write instructions i.e. OpCode=011, 110
MEMORY1:begin
//Read from mem peripheral and write to wdt
if(instr_decoder_output == 8'h01) begin
next_state = EXECUTE;
end else
next_state = curr_state.next;
end
MEMORY2:begin
next_state = curr_state.next;
end
WRITEBACK1:begin
next_state = curr_state.next;
end
WRITEBACK2:begin
readyToRW(!MWRITE, MWRITE);//Write new instruction to Instruction mem
next_state = IDLE;
end

//State for waiting for current instruction to finish
IDLE:begin
if(!CPUPREADY && !mem_cpu_ready)begin
next_state = curr_state;
end else begin
if(instr_decoder_output == 8'h06) begin
next_state = MEMORY1;
end else if (instr_decoder_output != 8'h06) begin
ce <= 0;
instr_decoder_enable <= 0;
next_state = curr_state.first;
end
end
end

//State to handle reset instructions
RESET:begin
ce <= 0;
instr_decoder_enable <= 0;
next_state = curr_state.first;
end
endcase
end

endmodule:cpu
```

2. CPU Testbench

```
module cpuTestBench;


```

```
bit clk = 0;
always begin
clk = ~clk;#5; //10ns Period
end

bit RUN;
bit CPUPREADY;
bit INCPURESET;
bit CPURESET;
logic[7:0] CPUSEL;
logic APBMASTERENABLE;
logic [7:0] addr;
logic [20:0] data;
logic CPUDONE;
logic CPUPERPHRESET;
logic [20:0] MPPRDATA;

int ploc = 0;
cpu dut (RUN, CPUPREADY, clk,
INCPURESET, APBMASTERENABLE, addr, data, CPUSEL, CPURESET,
CPUDONE,CPUPERPHRESET, MPPRDATA);

task cpu_reset;
@(posedge clk); #1;
CPURESET = 1;
@(posedge clk); #1;
CPURESET = 0;
@(posedge clk); #1;
endtask:cpu_reset
//Add an instruction to the main memory for the cpu to run
task ram_write(input logic [31:0] instr);
@(posedge clk); #1;
dut.imemory.mem[ploc] = instr;
RUN = 1;
CPUPREADY = 0;
ploc++;

@(posedge clk);
@(posedge clk);
@(posedge clk);
@(posedge clk);
@(posedge clk);
@(posedge clk);
@(posedge clk); #1;

CPUPREADY = 1;

wait(CPUDONE);
RUN = 0;
@(posedge clk);
endtask: ram_write;
//Add a main mem Write instr to the main memory
task ram_write_instr(input logic [31:0] write_instr, input logic [31:0]
new_instr); @(posedge clk); #1;
dut.imemory.mem[ploc] = write_instr;
ploc++;
dut.imemory.mem[ploc] = new_instr;
```

```
RUN = 1;
CPUPREADY = 0;
ploc++;

@(posedge clk);
@(posedge clk);
@(posedge clk);
@(posedge clk);
@(posedge clk); #1;

CPUPREADY = 1;

wait(CPUDONE);
RUN = 0;
@(posedge clk); #1;

endtask: ram_write_instr;

initial begin
cpu_reset;

for (int i=0; i<4096; i++) begin
dut.imemory.mem[i] = 0;
end

//Write

ram_write(32'b001_00010001_000000000000000000011);

ram_write(32'b100_00000001_000000000000000000011);

ram_write(32'b100_00000011_000000000000000000111);

ram_write(32'b100_00000101_000000000000000001111);
//Read

ram_write(32'b010_00010001_000000000000000000011);

ram_write(32'b101_00000001_000000000000000000011);

ram_write(32'b101_00000011_000000000000000000111);


ram_write(32'b101_00000101_000000000000000001111);

MPPRDATA = 3;

//New instruction mem rd wdt write

ram_write(32'b110_00100010_000000000000000000011);
```

```
ram_write_instr(32'b011_00000011_000000000000000000011,
32'b100_00010001_000000000000000001111);
ploc = 11;

ram_write(32'b000_00000001_000000000000000001111);

$finish;

end


endmodule
```

D. System Level Code

1. Testbenches

a. Random (system) Testbench

```
import typedef_pkg::environment;

program test(input logic clk, mem_intf intf);

//declaring environment instance
environment env;

initial begin
intf.run = 0;
//creating environment
env = new(intf);

//setting the repeat count of generator as 10, means to generate 10
packets
env.gen.repeat_count = 10;
env.gen.testcase_rand = 1; //Random Test
//calling run of env, it interns calls generator and driver
main tasks.
env.run();
//@(posedge clk); //Delay by a clk period to allow the last instruction to
        write to  cpu instruction mem
end
endprogram
```

b. Memory and WDT directed (system) testbench

```
import typedef_pkg::environment;
import typedef_pkg::transaction;

// System Test: Write data to memory module, use data in memory module to
   write to WDT  registers. Read back WDT register values.
program mem_and_wdt_test(input logic clk, mem_intf intf);
int counter = 0, running = 0;

class my_trans extends transaction;
function void pre_randomize();
// turn off randomization
```

```
//this.rand_mode(0);
opcode.rand_mode(0);
addr.rand_mode(0);
data.rand_mode(0);
opcode_2.rand_mode(0);
addr_2.rand_mode(0);
data_2.rand_mode(0);

// turn off constraints
//this.constraint_mode(0);
opcode_c.constraint_mode(0);
opcode_2_c.constraint_mode(0);
opcode_watchdog_c.constraint_mode(0);
opcode_2_watchdog_c.constraint_mode(0);
opcode_write_read_wdt_c.constraint_mode(0);
opcode_write_read_mem_c.constraint_mode(0);
opcode_instr_write_read_mem_c.constraint_mode(0);
//opcode_instr_2_write_read_mem_c.constraint_mode(0);

/* disabled constraints
cd_watchdog_c.constraint_mode(0);
cd_2_watchdog_c.constraint_mode(0);
kick_watchdog_c.constraint_mode(0);
kick_2_watchdog_c.constraint_mode(0);
*/

if (counter == 0) begin
// write to memory
opcode = 3'b100;
addr = 8'h05;
data = 21'h02;
// write to memory
opcode_2 = 3'b100;
addr_2 = 8'h06;
data_2 = 21'h0A;
end
else if (counter == 1) begin
// mem module read
opcode = 3'b101;
addr = 8'h05;
data = 21'h98;
// mem module read
opcode_2 = 3'b101;
addr_2 = 8'h06;
data_2 = 21'h99;
end
else if (counter == 2) begin
// memory module read to wdt write
opcode = 3'b110;
addr = 8'h11;
data = 21'h05; // lower 8 bits are the memory address to pull the data from
// memory module read to wdt write
opcode_2 = 3'b110;
addr_2 = 8'h33;
data_2 = 21'h06; // lower 8 bits are the memory address to pull the data
              from
end
```

```
else if (counter == 3) begin
// read from wdt
opcode = 3'b010;
addr = 8'h11;
data = 21'h77;
// read from wdt
opcode_2 = 3'b010;
addr_2 = 8'h33;
data_2 = 21'h88;
end

$display ("Counter =%0d", counter);
counter++;
//cnt++;
endfunction

//deep copy method
function my_trans do_copy();
my_trans trans;
trans = new();
trans.opcode = this.opcode;
trans.addr = this.addr;
trans.data = this.data;
trans.opcode_2 = this.opcode_2;
trans.addr_2 = this.addr_2;
trans.data_2 = this.data_2;
return trans;
endfunction
endclass

//declaring environment instance
environment env;
my_trans my_tr;

initial begin
intf.run = 0;
my_tr = new();
//creating environment
env = new(intf);

//setting the repeat count of generator as 10, means to generate 10 packets
env.gen.repeat_count = 4;
env.gen.testcase_rand = 0; //Manual Test
env.gen.trans = my_tr;
running = 1 ;
//calling run of env, it interns calls generator and driver main tasks.
env.run();
@(posedge clk); //Delay by a clk period to allow the last instruction to
          write to  cpu instruction mem
end
endprogram: mem_and_wdt_test

```

c. WDT directed (system) testbench

```
import typedef_pkg::environment;
import typedef_pkg::transaction;

// wdt system test
program wdt_test(input logic clk,
mem_intf intf);
int counter = 0,
running = 0;

class my_trans extends transaction;
function void pre_randomize();
// turn off randomization
//this.rand_mode(0);
opcode.rand_mode(0);
addr.rand_mode(0);
data.rand_mode(0);
opcode_2.rand_mode(0);
addr_2.rand_mode(0);
data_2.rand_mode(0);

// turn off constraints
//this.constraint_mode(0);
opcode_c.constraint_mode(0);
opcode_2_c.constraint_mode(0);
opcode_watchdog_c.constraint_mode(0);
opcode_2_watchdog_c.constraint_mode(0);
opcode_write_read_wdt_c.constraint_mode(0);
opcode_write_read_mem_c.constraint_mode(0);
opcode_instr_write_read_mem_c.constraint_mode(0);
//opcode_instr_2_write_read_mem_c.constraint_mode(0)
;
/* disabled constraints
cd_watchdog_c.constraint_mode(0);
cd_2_watchdog_c.constraint_mode(0);
kick_watchdog_c.constraint_mode(0);
kick_2_watchdog_c.constraint_mode(0)
;
*/

if (counter == 0) begin
// write to wdt
opcode = 3'b001;
addr = 8'h11;
data = 21'h02;
// write to wdt
opcode_2 = 3'b001;
addr_2 = 8'h33;
data_2 = 21'h0A;
end
else if (counter == 1) begin
// read from wdt
opcode = 3'b010;
addr = 8'h11;
data = 21'h88;
// read from wdt
opcode_2 = 3'b010;
addr_2 = 8'h33;
data_2 = 21'h99;
end
```

```
else if (counter == 2) begin
// write to wdt
opcode = 3'b001;
addr = 8'h22;
data = 21'h01;
// cpu reset
opcode_2 = 3'b000;
addr_2 = 8'h66;
data_2 = 21'h55;
end
else if (counter == 3) begin
// read from wdt
opcode = 3'b010;
addr = 8'h11;
data = 21'h77;
// read from wdt
opcode_2 = 3'b010;
addr_2 = 8'h33;
data_2 = 21'h88;
end

$display ("Counter =%0d", counter);
counter++;
//cnt++;
endfunction

//deep copy method
function my_trans do_copy();
my_trans trans;
trans = new();
trans.opcode = this.opcode;
trans.addr = this.addr;
trans.data = this.data;
trans.opcode_2 = this.opcode_2;
trans.addr_2 = this.addr_2;
trans.data_2 = this.data_2;
return trans;
endfunction
endclass

//declaring environment instance
environment env;
my_trans my_tr;

initial begin
intf.run = 0;
my_tr = new();
//creating environment
env = new(intf);

//setting the repeat count of generator as 10, means to generate 10
packets
env.gen.repeat_count = 4;
env.gen.testcase_rand = 0; //Manual Test
env.gen.trans = my_tr;
running = 1 ;
//calling run of env, it interns calls generator and driver main
tasks.
env.run();
```

```
@(posedge clk); //Delay by a clk period to allow the last instruction to write to  cpu instruction mem
end
endprogram: wdt_test


```

2. Other classes

```
package typedef_pkg;

typedef logic [2:0] opcode_t;
typedef logic [7:0] address_t;
typedef logic [20:0] data_t;
typedef logic [3:0] wait_t;

// Checker parameters //////////////////////////////////////////////////////
// cpu instructions use these addresses to set internal registers in the WDT
parameter CLOCK_DIVIDER_ADDR = 8'h11;
parameter KICK_ADDR = 8'h22;
parameter TIMEOUT_ADDR = 8'h33;

// internal register memory has a much lower, contiguous set of addresses
parameter INT_REG_CLK_DIV = 0;
parameter INT_REG_KICK = 1;
parameter INT_REG_TIMEOUT = 2;

parameter DEFAULT_CLK_DIV = 21'h0; // no clock division
parameter DEFAULT_TIMEOUT = 21'h19; // 25 cycles default
parameter DEFAULT_KICK = 21'h0;

//Class for randomized transactions
class transaction;
//declaring the transaction items
rand opcode_t opcode;
rand address_t addr;
rand data_t data;

// only used when opcode is a RAM Write 3'h011, because the
// 32bit instruction to write to instruction memory is needed
rand opcode_t opcode_2;
rand address_t addr_2;
rand data_t data_2;

// wdt related
data_t prdata_0;

// memory module related
data_t prdata_1;

logic [1:0] cnt; // remove later if unused

//constaint, limit opcode as some are unused
```

```
constraint opcode_c { opcode inside {[3'b001:3'b101]}; };
constraint opcode_2_c { opcode_2 inside {[3'b001:3'b101]}; };

// watchdog instruction only has 3 specific non contiguous addresses
constraint opcode_watchdog_c { opcode inside{3'b001, 3'b010} -> addr inside{8'h11, 8'h22, 8'h33, 8'h44};};
constraint opcode_2_watchdog_c { opcode_2 inside{3'b001, 3'b010} ->
        addr_2 inside{ 8'h11, 8'h22, 8'h33, 8'h44};};

//Costraints for write then read
constraint opcode_write_read_wdt_c { opcode == 3'b001 -> opcode_2
        inside{3'b010}; addr_2 inside{addr};};
constraint opcode_write_read_mem_c { opcode == 3'b100 -> opcode_2
        inside{3'b101}; addr_2 inside{addr};};

constraint opcode_instr_write_read_mem_c { opcode == 3'b011 -> addr > 50;};
constraint opcode_2_instr_write_read_mem_c { opcode_2 == 3'b011 -> addr_2 > 50;};



/* constraint cd_watchdog_c { addr == 8'h11 -> data inside{[21'h05:21'h01]};}
constraint cd_2_watchdog_c { addr_2 == 8'h11 -> data_2 inside {[21'h05:21'h01]};};

constraint kick_watchdog_c { addr == 8'h22 -> data inside {21'h00, 21'h01}; }
constraint kick_2_watchdog_c { addr_2 == 8'h22 -> data_2 inside {21'h00, 21'h01};  } */

function transaction do_copy();
transaction trans;
trans = new();
trans.opcode = this.opcode;
trans.addr = this.addr;
trans.data = this.data;
trans.opcode_2 = this.opcode_2;
trans.addr_2 = this.addr_2;
trans.data_2 = this.data_2;
return trans;
endfunction
endclass : transaction

//Generator class
class generator;
//declaring transaction class
rand transaction trans;
int testcase_rand;
//declaring mailbox
mailbox gen2driv;

//repeat count, to specify number of items to generate
int repeat_count;

//event
event ended;

//constructor
```

```
function new(input mailbox gen2driv,input event ended);
//getting the mailbox handle from env
this.gen2driv = gen2driv;
this.ended = ended;
endfunction

//main task, generates(create and randomizes) the repeat_count number of  transaction packets and puts into mailbox
task main();
repeat(repeat_count) begin
if(testcase_rand)begin
trans = new();
end
if( !trans.randomize() ) $fatal(0, "Gen:: trans randomization failed");
gen2driv.put(trans.do_copy());
end
-> ended;
endtask
endclass : generator

//Driver class
class driver;
//used to count the number of transactions
int no_transactions;
//creating virtual interface handle
virtual mem_intf mem_vif;
//creating mailbox handle
mailbox gen2driv;
//constructor
function new(virtual mem_intf mem_vif,mailbox gen2driv);
//getting the interface
this.mem_vif = mem_vif;
//getting the mailbox handle from environment
this.gen2driv = gen2driv;
endfunction
//Adding a reset task, which initializes the Interface signals to default values
//For simplicity, define is used to access interface signals.
//`define DRIV_IF mem_vif.DRIVER.driver_cb
`define DRIV_IF mem_vif.driver_cb
//`DRIV_IF will point to mem_vif.DRIVER.driver_cb

//Reset task, Reset the Interface signals to default/initial values
task reset;
wait(mem_vif.reset);
$display("--------- [DRIVER] Reset Started ---------");
`DRIV_IF.opcode <= 0;
`DRIV_IF.addr <= 0;
`DRIV_IF.data <= 0;
`DRIV_IF.opcode_2 <= 0;
`DRIV_IF.addr_2 <= 0;
`DRIV_IF.data_2 <= 0;
wait(!mem_vif.reset);
$display("--------- [DRIVER] Reset Ended---------");
@(posedge mem_vif.clk); @(posedge mem_vif.clk);
endtask
//drive the transaction items to interface signals
```

```
task drive;
forever begin
transaction trans;
gen2driv.get(trans);

$display("--------- [DRIVER-TRANSFER: %0d] ---------",no_transactions);
@(posedge mem_vif.clk);
`DRIV_IF.opcode <= trans.opcode;
`DRIV_IF.addr <= trans.addr;
`DRIV_IF.data <= trans.data;
`DRIV_IF.opcode_2 <= trans.opcode_2;
`DRIV_IF.addr_2 <= trans.addr_2;
`DRIV_IF.data_2 <= trans.data_2;

// instruction takes 4 cycles to queue inst, 2 cycles for APB master to execute, read prdata 1 cycle later

mem_vif.run = 1;
wait(mem_vif.cpudone);
@(posedge mem_vif.clk);
wait(!mem_vif.cpudone);
wait(mem_vif.cpudone);
//wait(!mem_vif.cpudone);

// if(trans.opcode != 3'b011) begin
// wait(mem_vif.cpudone);
mem_vif.run = 0;
// if(trans.opcode_2 == 3'b010 || trans.opcode_2 == 3'b101) begin
// wait(mem_vif.prdata_0 || mem_vif.prdata_1);
// end
// end
@(posedge mem_vif.clk); @(posedge mem_vif.clk);
@(posedge mem_vif.clk); @(posedge mem_vif.clk);
@(posedge mem_vif.clk); @(posedge mem_vif.clk);


// if watch dog write or read opcodes
if (trans.opcode == 3'b001 || trans.opcode == 3'b010) begin
trans.prdata_0 = `DRIV_IF.prdata_0;
end
//Instr mem write
else if(trans.opcode == 3'b011) begin
end
//Mem peripheral write or read
else if (trans.opcode == 3'b100 || trans.opcode == 3'b101) begin
trans.prdata_1 = `DRIV_IF.prdata_1;
end

if (trans.opcode == 3'b001 || trans.opcode == 3'b010) begin
$display("[DRIVER] opcode = %0h \taddr = %0h \tdata = %0h \tprdata = %0h", trans.opcode, trans.addr, trans.data, `DRIV_IF.prdata_0);
$display("[DRIVER] opcode_2 = %0h \taddr_2 = %0h \tdata_2 = %0h \tprdata = %0h", trans.opcode_2, trans.addr_2, trans.data_2, `DRIV_IF.prdata_0);
end
else if (trans.opcode == 3'b100 || trans.opcode == 3'b101)
$display("[DRIVER] opcode = %0h \taddr = %0h \tdata = %0h \tprdata = %0h",
        trans.opcode, trans.addr, trans.data, `DRIV_IF.prdata_1);

if(trans.opcode == 3'b011)begin
$display("[DRIVER] opcode2 = %0h \taddr2 = %0h \tdata2 = %0h",
           trans. opcode_2, trans.addr_2, trans.data_2);
end

$display("--------- [DRIVER-TRANSFER DONE: %0d]
---------",no_transactions);
no_transactions++;
end
endtask
//
task main;
forever begin
fork
//Thread-1: Waiting for reset
begin
wait(mem_vif.reset);
end
//Thread-2: Calling drive task
begin
forever
drive();
end
join_any
disable fork;
end
endtask
endclass : driver

//Monitor class
class monitor;
`define MON_IF mem_vif.monitor_cb
//creating virtual interface handle
virtual mem_intf mem_vif;

//creating mailbox handle
mailbox mon2scb;

//Event for monitor to scoreboard
event mon2scbhandshake;
event scb2monhandshake;

//constructor
function new(virtual mem_intf mem_vif, mailbox mon2scb, input event
    mon2scbhandshake, input event scb2monhandshake);
//getting the interface
this.mem_vif = mem_vif;
//getting the mailbox handles from environment
this.mon2scb = mon2scb;

this.mon2scbhandshake = mon2scbhandshake;
this.scb2monhandshake = scb2monhandshake;
endfunction

//Samples the interface signal and send the sample packet to
scoreboard
task main;
```

```
forever begin
transaction trans;
trans = new();

wait(mem_vif.cpudone);
wait(!mem_vif.cpudone);
wait(mem_vif.cpudone);
// wait(mem_vif.prdata_0 || mem_vif.prdata_1);
@(posedge mem_vif.clk);
//@(posedge mem_vif.clk);

trans.opcode = `MON_IF.opcode;
trans.addr = `MON_IF.addr;
trans.data = `MON_IF.data;
trans.opcode_2 = `MON_IF.opcode_2;
trans.addr_2 = `MON_IF.addr_2;
trans.data_2 = `MON_IF.data_2;
trans.prdata_0 = `MON_IF.prdata_0;
trans.prdata_1 = `MON_IF.prdata_1;

mon2scb.put(trans);
->mon2scbhandshake;
@scb2monhandshake;
end
endtask

endclass : monitor

//Scoreboard class
class scoreboard;
//creating mailbox handle
mailbox mon2scb;

data_t [0:3] mem_0;
data_t [0:255] mem_1;
address_t new_addr; // used for wdt internal register addres
conversion

//used to count the number of transactions
int no_transactions;

//Event for proper timing with monitor
event mon2scbhandshake;
event scb2monhandshake;

//constructor
function new(mailbox mon2scb, input event mon2scbhandshake,
    input event scb2monhandshake);
//getting the mailbox handles from environment
this.mon2scb = mon2scb;
this.mon2scbhandshake = mon2scbhandshake;
this.scb2monhandshake = scb2monhandshake;
//foreach(mem_0[i]) mem_0[i] = 0;
foreach(mem_1[i]) mem_1[i] = 0;
mem_0[INT_REG_CLK_DIV] = DEFAULT_CLK_DIV;
mem_0[INT_REG_TIMEOUT] = DEFAULT_TIMEOUT;
mem_0[INT_REG_KICK] = DEFAULT_KICK;
mem_0[3] = 0;
```

```
endfunction

//stores the parts of the instructions
task main;
transaction trans;

forever begin
//#50;
@mon2scbhandshake;
mon2scb.get(trans);

// if watchdog related opcode, convert special addr to proper array
index if (trans.opcode == 3'b010 || trans.opcode == 3'b001) begin
if (trans.addr == CLOCK_DIVIDER_ADDR)
new_addr = INT_REG_CLK_DIV;
else if (trans.addr == KICK_ADDR)
new_addr = INT_REG_KICK;
else if (trans.addr == TIMEOUT_ADDR)
new_addr = INT_REG_TIMEOUT;
else
new_addr = 3;
end

if((trans.opcode == 3'b000 || trans.opcode_2 == 3'b000) && trans.opcode !=
          3'b011) begin
$display("[SCB-PASS-RESET] CPU RESET");
end

// simulate write for wdt
if (trans.opcode == 3'b001) begin
mem_0[new_addr] = trans.data;
mem_0[3] = 0;
$display("[SCB-PASS] WDT Write:: Addr = %0h,\t Data = %0h", trans.addr,
          trans. data);

// check read for wdt
if (trans.opcode_2 == 3'b010) begin
if (mem_0[new_addr] != trans.prdata_0)
 $error("[CHK-FAIL] Addr = %0h,\t Data :: Expected = %0h Actual = %0h",
              trans.addr_2,mem_0[new_addr],trans.prdata_0);
else
 $display("[CHK-PASS] Addr = %0h,\t Data :: Expected = %0h Actual = %0h",
              trans.addr_2,mem_0[new_addr],trans.prdata_0);
end
end

// simulate write for memory
if (trans.opcode == 3'b100) begin
mem_1[trans.addr] = trans.data;
$display("[SCB-PASS] MemPer Write:: Addr = %0h,\t Data = %0h", trans.addr,
          trans .data);

if (trans.opcode_2 == 3'b101) begin
if (mem_1[trans.addr] != trans.prdata_1)
 $error("[CHK-FAIL] Addr = %0h,\t Data :: Expected = %0h Actual = %0h",
              trans.addr,mem_1[trans.addr],trans.prdata_1);
```

```
else
 $display("[CHK-PASS] Addr = %0h,\t Data :: Expected = %0h Actual = %0h",
                trans.addr,mem_1[trans.addr],trans.prdata_1);
end
end

// check read for wdt
if (trans.opcode == 3'b010) begin
if (mem_0[new_addr] != trans.prdata_0)
 $error("[CHK-FAIL - INSTR 1] Addr = %0h,\t Data :: Expected = %0h Actual
            =  %0h",trans.addr,mem_0[new_addr],trans.prdata_0);
else
 $display("[CHK-PASS - INSTR 1] Addr = %0h,\t Data :: Expected = %0h Actual
             =  %0h",trans.addr,mem_0[new_addr],trans.prdata_0);
end


// check read for memory
if (trans.opcode == 3'b101) begin
if (mem_1[trans.addr] != trans.prdata_1)
 $error("[CHK-FAIL - INSTR 1] Addr = %0h,\t Data :: Expected = %0h Actual
           =  %0h",trans.addr,mem_1[trans.addr],trans.prdata_1);
else
 $display("[CHK-PASS - INSTR 1] Addr = %0h,\t Data :: Expected = %0h Actual
            =  %0h",trans.addr,mem_1[trans.addr],trans.prdata_1);
end

if (trans.opcode_2 == 3'b010 || trans.opcode_2 ==
3'b001) begin
if (trans.addr_2 == CLOCK_DIVIDER_ADDR)
new_addr = INT_REG_CLK_DIV;
else if (trans.addr_2 == KICK_ADDR)
new_addr = INT_REG_KICK;
else if (trans.addr_2 == TIMEOUT_ADDR)
new_addr = INT_REG_TIMEOUT;
else
new_addr = 3;
end


if (trans.opcode_2 == 3'b010 && trans.opcode != 3'b001 && trans.opcode !=
           3'b011) begin
if (mem_0[new_addr] != trans.prdata_0)
 $error("[CHK-FAIL - INSTR 2] Addr = %0h,\t Data :: Expected = %0h Actual
           =  %0h",trans.addr_2,mem_0[new_addr],trans.prdata_0);
else
 $display("[CHK-PASS - INSTR 2] Addr = %0h,\t Data :: Expected = %0h Actual
            =  %0h",trans.addr_2,mem_0[new_addr],trans.prdata_0);
end



if (trans.opcode_2 == 3'b101 && trans.opcode != 3'b100 && trans.opcode !=
             3'b011) begin
if (mem_1[trans.addr] != trans.prdata_1)
 $error("[CHK-FAIL - INSTR 2] Addr = %0h,\t Data :: Expected = %0h Actual
          =  %0h",trans.addr_2,mem_1[trans.addr],trans.prdata_1);
```

```
else
 $display("[CHK-PASS - INSTR 2] Addr = %0h,\t Data :: Expected = %0h Actual
            =  %0h",trans.addr_2,mem_1[trans.addr],trans.prdata_1);
end

// another wdt write
if(trans.opcode_2 == 3'b001 && trans.opcode !=
3'b011) begin mem_0[new_addr] = trans.data_2;
mem_0[3] = 0;
$display("[SCB-PASS - INSTR 2] WDT Write:: Addr = %0h,\t Data = %0h",
              trans. addr_2, trans.data_2);
end

// another memory module write
if (trans.opcode_2 == 3'b100 && trans.opcode !=
3'b011) begin mem_1[trans.addr] = trans.data_2;
$display("[SCB-PASS - INSTR 2] MemPer Write:: Addr = %0h,\t Data = %0h",
              trans. addr, trans.data);
end

if(trans.opcode == 3'b011) begin
$display("[SCB-PASS] CPU INSTR MEM Write");
end


no_transactions++;
->scb2monhandshake;
end
endtask
endclass : scoreboard


//Enviornment Class
class environment;
//generator and driver instance
generator gen;
driver driv;
monitor mon;
scoreboard scb;

//mailbox handle's
mailbox gen2driv;
mailbox mon2scb;

//event for synchronization between generator and test
event gen_ended;
event mon2scbhandshake;
event scb2monhandshake;

//virtual interface
virtual mem_intf mem_vif;

//constructor
function new(virtual mem_intf mem_vif);
//get the interface from test
this.mem_vif = mem_vif;
```

```

//creating the mailbox (Same handle will be shared across generator and driver)
gen2driv = new();
mon2scb = new();

//creating generator and driver
gen = new(gen2driv,gen_ended);
driv = new(mem_vif,gen2driv);
mon = new(mem_vif,mon2scb, mon2scbhandshake, scb2monhandshake);
scb = new(mon2scb, mon2scbhandshake, scb2monhandshake);
endfunction

//
task pre_test();
driv.reset();
endtask

task test();
fork
gen.main();
driv.main();

mon.main();
scb.main();
join_any
endtask

task post_test();
wait(gen_ended.triggered);
wait(gen.repeat_count == driv.no_transactions);
wait(gen.repeat_count == scb.no_transactions);
endtask

//run task
task run;
pre_test();
test();
post_test();
//$finish;
endtask

endclass : environment

endpackage : typedef_pkg
```

```
//`timescale 1ns / 1ps

`include "mem_intf.sv"

module top;
parameter PERIOD = 10, HALFPERIOD = PERIOD/2, TIMEDELAY = 1;

logic clk = 0, reset = 0;
logic [31:0] write_instr, new_instr;

//clock generation
always begin
clk = ~clk; #(HALFPERIOD);
end

int ploc = 0, transcnt = 0;

always_comb begin
write_instr = 0;
new_instr = 0;
if((intf.opcode || intf.addr || intf.data)) begin
$display("\tOpcode: %b, addr:%b, data: %b", intf.opcode, intf.addr, intf.data);
// if(intf.opcode == 3'b011) begin //Instruction memory write Instruction, Opcode 3'b011
$display("\tOpcode2: %b, addr2:%b, data2: %b", intf.opcode_2, intf.addr_2, intf.data_2);
// end

//Concate random var into single aarray and write instruc to cpu instruc mem
write_instr = {intf.opcode, intf.addr, intf.data};
new_instr = {intf.opcode_2, intf.addr_2, intf.data_2};
ram_write_instr(write_instr, new_instr);

$display(" *** [CPU INSTR-WROTE %0d] ***",transcnt-1);
end
end

//reset Generation
task subSystemReset;
@(posedge clk)#TIMEDELAY;
reset = 1;
// foreach(dut.cpu1.imemory.mem[i]) dut.cpu1.imemory.mem[i] = 0;
@(posedge clk)#TIMEDELAY;
reset = 0;
@(posedge clk)
@(posedge clk);
@(posedge clk);
endtask:subSystemReset

//Write Instruction to Cpu's Instruction mem
task ram_write_instr(input logic [31:0] write_instr, input logic [31:0] new_instr);
//@(posedge clk); #1;
dut.cpu1.imemory.mem[ploc] = write_instr;
ploc++;
transcnt++;
dut.cpu1.imemory.mem[ploc] = new_instr;
ploc++;
/*if(write_instr[31:29] == 3'b011) begin //Instruction memory write  Instruction, Opcode 3'b011
dut.cpu1.imemory.mem[ploc] = new_instr;
ploc++;
end else begin
dut.cpu1.imemory.mem[ploc] = 0;
```

```
end*/
// @(posedge clk); #1;
endtask: ram_write_instr;



//initialization init(clk, reset);// prework stuff fill cpu memory, generatore  clock, reset, etc

// instantiate the interface that passes signals from testbench and DUT
mem_intf intf(clk,reset);

DUT dut(clk, reset, intf.run, intf.prdata_0, intf.prdata_1, intf.wdt_trigger_reset, intf.cpudone); // has the design files

test t1(clk, intf); // holds the environment (gen/driv/etc)

//wdt_test t2 (clk, intf);

//mem_and_wdt_test t3 (clk, intf);

initial begin
subSystemReset;

//enabling the wave dump
// $dumpfile("dump.vcd"); $dumpvars;
end
endmodule : top
```

```
`include "cpu_if.sv"
`include "sys_if.sv"

module DUT(input logic clk,
input logic reset,
input logic run,
output logic [20:0] prdata_0,
output logic [20:0] prdata_1,
output logic wdt_trigger_reset,
output logic cpuinstrdone);

//clock and reset signal declaration
logic
wren_w,wren_m, //Watchdog r/w signals
rden_w, rden_m,//Mem per r/w signals
ce,
incpureset, // signal from watchdog timer to CPU to force cpu reset
cpudone, //signal from cpu when instruction is completed
cpuperphreset; // signal from cpu driven to peripherals to force a  peripheral reset

logic [20:0] prdata_master_0, prdata_master_1;

//instantiate interfaces
cpu_if cpuif(clk);
sys_if sysif(clk);

//Push output wire signals
```

```
assign prdata_0 = sysif.prdata_0;
assign prdata_1 = sysif.prdata_1;
assign wdt_trigger_reset = incpureset;
assign cpuinstrdone = cpudone;

// APB interfaces for cpu and peripherals
apbMaster apb_master (.APBMASTERENABLE (cpuif.APBMASTERENABLE),
.CPUSEL (cpuif.CPUSEL), .addr (cpuif.addr),
.data (cpuif.data),
.PCLK (sysif.pclk),
.PRESET (cpuperphreset), .PSEL_0 (sysif.psel_0),
.PSEL_1 (sysif.psel_1), .PENABLE (sysif.penable),
.PADDR (sysif.paddr),
.PWRITE (sysif.pwrite), .PWDATA (sysif.pwdata),
.CPUPREADY (cpuif.CPUREADY), .PREADY_0 (sysif.pready_0),
.PREADY_1 (sysif.pready_1), .PRDATA_1
(prdata_master_1), .PRDATA_1_OUT (cpuif.MPPRDATA) );

// rename APB_Slave_WDT to generic APB_Slave
APB_Slave_WDT apb_memory (.pclk(sysif.pclk),
.presetn(cpuperphreset),
.paddr(sysif.paddr),
.psel(sysif.psel_1),
.penable(sysif.penable),
.pwrite(sysif.pwrite),
.pwdata(sysif.pwdata),
.wait_time(sysif.wait_time),
.cpu_forced_reset(cpuperphreset),
.prdata(sysif.prdata_1),
.pready(sysif.pready_1),
.wren(wren_m),
.rden(rden_m),
.prdata_out(prdata_master_1),
.ce(sysif.ce_1));

APB_Slave_WDT apb_wdt
(.pclk(sysif.pclk),
.presetn(cpuperphreset),
.paddr(sysif.paddr),
.psel(sysif.psel_0),
.penable(sysif.penable),
.pwrite(sysif.pwrite),
.pwdata(sysif.pwdata),
.wait_time(sysif.wait_time),
.cpu_forced_reset (cpuperphreset),
.prdata(sysif.prdata_0),
.pready(sysif.pready_0),
.wren(wren_w),
.rden(rden_w),
.prdata_out(prdata_master_0),
.ce(sysif.ce_0));

// cpu and peripheral modules
cpu cpu1( .RUN(run),
.CPUPREADY(cpuif.CPUREADY),
.CCLK(cpuif.clk),
.INCPURESET(incpureset),
.APBMASTERENABLE(cpuif.APBMASTERENABLE),
.addr(cpuif.addr),
.data(cpuif.data),
.CPUSEL(cpuif.CPUSEL),
.CPURESET(reset),
.CPUDONE(cpudone),
.CPUPERPHRESET(cpuperphreset),
.MPPRDATA(cpuif.MPPRDATA));

WDT wdt1(.pclk(sysif.pclk),
.paddr(sysif.paddr),
.pwdata(sysif.pwdata),
.wren(wren_w),
.rden(rden_w),
.prdata(sysif.prdata_0),
.cpu_forced_reset(cpuperphreset),
.cpu_reset_trig(incpureset));

memory mem1 (.clk (sysif.pclk),
.addr (sysif.paddr),
.ce (sysif.ce_1),
.wren (wren_m),
.rden (rden_m),
.wr_data (sysif.pwdata),
.rd_data (sysif.prdata_1));

endmodule: DUT

```

3. Interfaces

```
// import path for package files
import typedef_pkg::opcode_t;
import typedef_pkg::address_t;
import typedef_pkg::data_t;

interface mem_intf(input logic clk,reset);

opcode_t opcode;
address_t addr;
data_t data;

// only used when opcode is a RAM Write 3'h011, because the
// 32bit instruction to write to instruction memory is needed
opcode_t opcode_2;
address_t addr_2;
data_t data_2;

// wdt related
data_t prdata_0;
logic wdt_trigger_reset;

// memory module related
data_t prdata_1;
```

```

//cpu related
logic cpudone;

// used to start cpu running after cpu main memory is filled with instructions
logic run;


//driver clocking block
clocking driver_cb @(posedge clk);
default input #1 output #1;
output opcode;
output addr;
output data;
output opcode_2;
output addr_2;
output data_2;
input prdata_0;
input wdt_trigger_reset;
input prdata_1;
endclocking

//monitor clocking block
clocking monitor_cb @(posedge clk);
default input #1 output #1;
input opcode;
input addr;
input data;
input opcode_2;
input addr_2;
input data_2;
input prdata_0;
input wdt_trigger_reset;
input prdata_1;
endclocking


//driver modport
modport DRIVER (clocking driver_cb,input clk,reset);

//monitor modport
modport MONITOR (clocking monitor_cb,input clk,reset);

endinterface: mem_intf
```

```
// import path for package files
import typedef_pkg::address_t;
import typedef_pkg::data_t;
import typedef_pkg::wait_t;

interface sys_if(input bit pclk);
// declare signals
logic psel_0, psel_1,
presetn, penable, pwrite, pready_0, pready_1, ce_0, ce_1;

address_t paddr;
```

```
data_t pwdata, prdata_0, prdata_1;
wait_t wait_time = 0;

// DUT modport could be used if creating a new slave DUT or modifying and
      existing  one
//modport WDT (input pclk, paddr, pwdata,
// output prdata);

modport WDT_APBSLAVE (input pclk, presetn, psel_0, penable, pwrite,
    paddr, pwdata, wait_time,
output prdata_0, pready_0);

modport memory_APBSLAVE (input pclk, presetn, psel_1, penable, pwrite,
    paddr, pwdata, wait_time,
output prdata_1, pready_1);

endinterface : sys_if


```

```
// import path for package files
import typedef_pkg::address_t;
import typedef_pkg::data_t;

interface cpu_if(input logic clk);
// declare signals
logic APBMASTERENABLE, CPUREADY;
address_t CPUSEL, addr;
data_t data;

data_t MPPRDATA;

// DUT modport could be used if creating a new slave DUT or modifying and
      existing  one
modport CPU (input clk, CPUREADY,
output APBMASTERENABLE, addr, data, CPUSEL);

modport APBMASTER (input clk, CPUREADY, APBMASTERENABLE, addr,
data, CPUSEL);
endinterface: cpu_if

```