8-Bit Pipelined Picoblaze Validation Plan

ECE-571 Fall

Aalap Khanolkar, aalap@pdx.edu

Kathyayani Neerudi, kneerudi@pdx.edu

Niko Nikolov, nnikolov@pdx.edu

Date 11-26-22

Table of contents

1. Introduction

In brief, coverage includes three components:
(1) Code Coverage (which is structural) which needs to be 100 % .
(2) Functional Coverage that need to be designed to cover functionality (i.e. intent) of the entire design and must completely cover the design.
(3) Temporal domain coverage (using SVA 'cover' feature) which need to be carefully designed to fully cover all required temporal domain conditions
of the design. [1]

Traditional verification can be called Black Box verification with Black Box observability, meaning, you apply vectors/transactions at the primary input of the 'block' without caring for what's in the block (blackbox verification) and you observe the behavior of the block only at the primary outputs (blackbox observability). Assertions on the other hand allow you to do black box verification with white box (internal to the block) observability. [1]

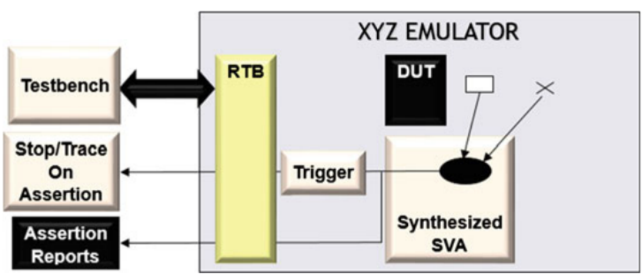


Figure: Assertions for HW emulation [1]

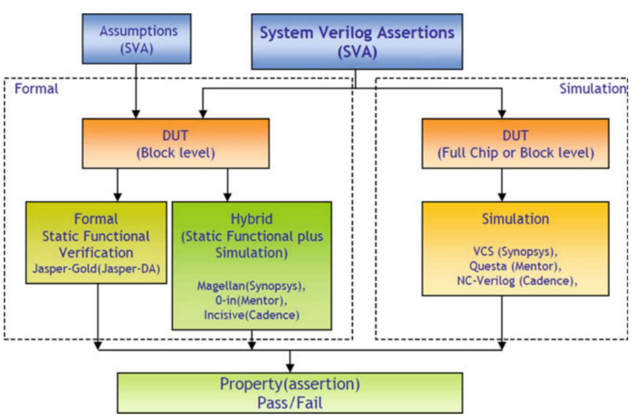


Figure: Static Assertions for HW emulation [1]

2. Design Description

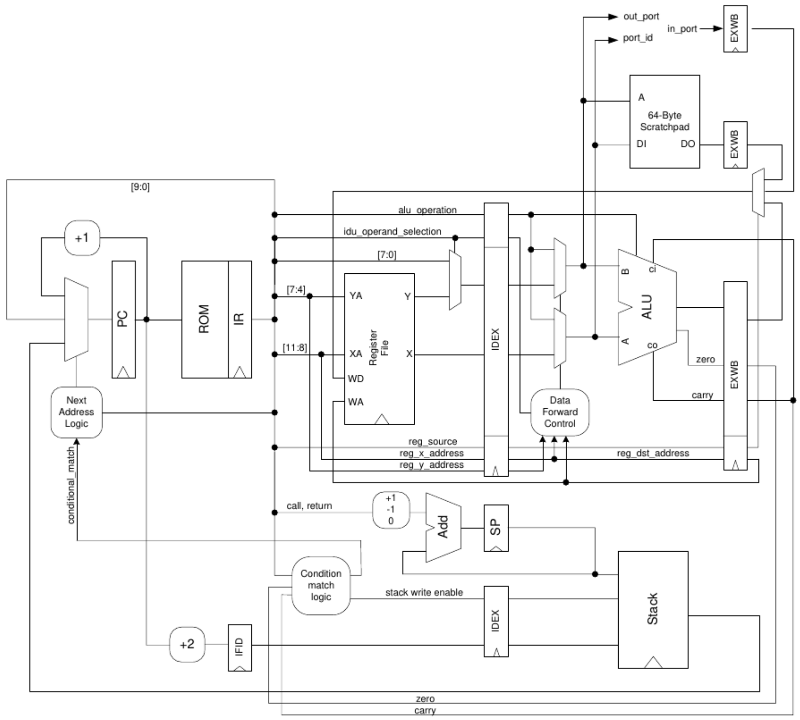
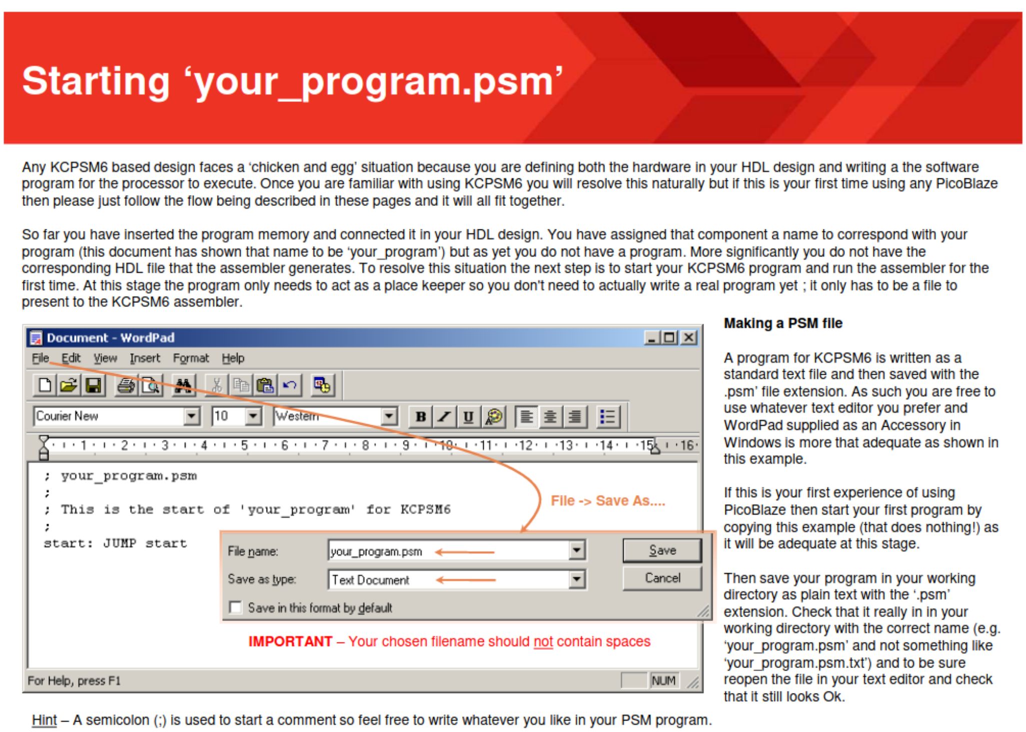


Figure: Picoblaze block diagram



Starting 'your_program.psm'

Any KCPSM6 based design faces a 'chicken and egg' situation because you are defining both the hardware in your HDL design and writing a the software program for the processor to execute. Once you are familiar with using KCPSM6 you will resolve this naturally but if this is your first time using any PicoBlaze then please just follow the flow being described in these pages and it will all fit together.

So far you have inserted the program memory and connected it in your HDL design. You have assigned that component a name to correspond with your program (this document has shown that name to be 'your_program') but as yet you do not have a program. More significantly you do not have the corresponding HDL file that the assembler generates. To resolve this situation the next step is to start your KCPSM6 program and run the assembler for the first time. At this stage the program only needs to act as a place keeper so you don't need to actually write a real program yet ; it only has to be a file to present to the KCPSM6 assembler.

Making a PSM file

A program for KCPSM6 is written as a standard text file and then saved with the .psm' file extension. As such you are free to use whatever text editor you prefer and WordPad supplied as an Accessory in Windows is more that adequate as shown in this example.

If this is your first experience of using PicoBlaze then start your first program by copying this example (that does nothing!) as it will be adequate at this stage.

Then save your program in your working directory as plain text with the '.psm' extension. Check that it really in in your working directory with the correct name (e.g. 'your_program.psm' and not something like 'your_program.psm.txt') and to be sure reopen the file in your text editor and check that it still looks Ok.

Hint – A semicolon (;) is used to start a comment so feel free to write whatever you like in your PSM program.

Figure: Creating PSM file

Instruction	Description	Function	ZERO	CARRY
ADD sX, kk	Add register `sX` with literal `kk`	sX ← sX + kk	?	?
ADD sX, sY	Add register `sX` with register `sY`	sX ← sX + sY	?	?
ADDCY sX, kk (ADDC)	Add register `sX` with literal `kk` with CARRY bit	sX ← sX + kk + CARRY	?	?
ADDCY sX, sY (ADDC)	Add register `sX` with register `sY` with CARRY bit	sX ← sX + sY + CARRY	?	?
AND sX, kk	Bitwise AND register `sX` with literal `kk`	sX ← sX AND kk	?	0
AND sX, sY	Bitwise AND register `sX` with register `sY`	sX ← sX AND sY	?	0
CALL aaa	Unconditionally call subroutine at aaa	TOS ← PC PC ← aaa	-	-
CALL C, aaa	If CARRY flag set, call subroutine at aaa	If CARRY=1, {TOS ← PC, PC ← aaa}	-	-
CALL NC, aaa	If CARRY flag not set, call subroutine at aaa	If CARRY=0, {TOS ← PC, PC ← aaa}	-	-
CALL NZ, aaa	If ZERO flag not set, call subroutine at aaa	If ZERO=0, {TOS ← PC, PC ← aaa}	-	-
CALL Z, aaa	If ZERO flag set, call subroutine at aaa	If ZERO=1, {TOS ← PC, PC ← aaa}	-	-
COMPARE sX, kk (COMP)	Compare register `sX` with literal `kk`. Set CARRY and ZERO flags as appropriate. Registers are unaffected.	If sX=kk, ZERO ← 1 If sX<kk, CARRY ← 1	?	?
COMPARE sX, sY (COMP)	Compare register `sX` with register `sY`. Set CARRY and ZERO flags as appropriate. Registers are unaffected.	If sX=sY, ZERO ← 1 If sX<sY, CARRY ← 1	?	?
DISABLE INTERRUPT (DINT)	Disable interrupt input	INTERRUPT_ENABLE ← 0	-	-

Figure: Picoblaze ISA

Instruction	Description	Function	ZERO	CARRY
ADD sX, kk	Add register `sX` with literal `kk`	sX ← sX + kk	?	?
ADD sX, sY	Add register `sX` with register `sY`	sX ← sX + sY	?	?
ADDCY sX, kk (ADDC)	Add register `sX` with literal `kk` with CARRY bit	sX ← sX + kk + CARRY	?	?
ADDCY sX, sY (ADDC)	Add register `sX` with register `sY` with CARRY bit	sX ← sX + sY + CARRY	?	?
AND sX, kk	Bitwise AND register `sX` with literal `kk`	sX ← sX AND kk	?	0
AND sX, sY	Bitwise AND register `sX` with register `sY`	sX ← sX AND sY	?	0
CALL aaa	Unconditionally call subroutine at aaa	TOS ← PC PC ← aaa	-	-
CALL C, aaa	If CARRY flag set, call subroutine at aaa	If CARRY=1, {TOS ← PC, PC ← aaa}	-	-
CALL NC, aaa	If CARRY flag not set, call subroutine at aaa	If CARRY=0, {TOS ← PC, PC ← aaa}	-	-
CALL NZ, aaa	If ZERO flag not set, call subroutine at aaa	If ZERO=0, {TOS ← PC, PC ← aaa}	-	-
CALL Z, aaa	If ZERO flag set, call subroutine at aaa	If ZERO=1, {TOS ← PC, PC ← aaa}	-	-
COMPARE sX, kk (COMP)	Compare register `sX` with literal `kk`. Set CARRY and ZERO flags as appropriate. Registers are unaffected.	If sX=kk, ZERO ← 1 If sX<kk, CARRY ← 1	?	?
COMPARE sX, sY (COMP)	Compare register `sX` with register `sY`. Set CARRY and ZERO flags as appropriate. Registers are unaffected.	If sX=sY, ZERO ← 1 If sX<sY, CARRY ← 1	?	?
DISABLE INTERRUPT (DINT)	Disable interrupt input	INTERRUPT_ENABLE ← 0	-	-

Figure: Picoblaze ISA

Figure: Picoblaze ISA

Address Space	Size (Depth x Width)	Addressing Modes	Instructions that Operate on Address Space
Instruction	1Kx18	Direct	• `JUMP` • `CALL` • `RETURN` • `RETURNI` • INTERRUPT event • RESET event All others increment the PC to the next location
Register File	16x8	Direct	• `LOAD` • `AND` • `OR` • `XOR` • `TEST` (read only) • `ADD` • `ADDCY` • `SUB` • `SUBCY` • `COMPARE` (read only) • `SR0` • `SR1` • `SRX` • `SRA` • `RR` • `SL0` • `SL1` • `SLX` • `SLA` • `RL` • `INPUT` • `OUTPUT` (read only) • `STORE` (read only) • `FETCH`
Scratchpad RAM	64x8	Direct Indirect	• `STORE` • `FETCH`
I/O	256x8	Direct Indirect	• `INPUT` • `OUTPUT`
CALL/RETURN Stack	31x10	N/A	• `CALL` • Enabled INTERRUPT event • `RETURN` • `RETURNI` • RESET event

Figure: Picoblaze Address Space

3. Coverage

At the end we are targeting above 90% coverage.

4. Assertions

We will use System Verilog Assertion in our design.

5. Unit Level Testing Plans

5.1. Instruction Fetch

The test cases created for instruction fetch will mainly verify that the instructions are correctly fetched and also check if the program counter jumps to the target or increments, when the branch is taken or not taken respectively. Finally, we also check for rollover condition where the PC should rollback to the starting address when it goes out of bound. Assertions are created using expected outputs that are obtained from the register.

Team Member	Design Unit	Test Name
Kathy	IF	Reset -> Reset PC to 0, disable interrupts and clear flags
Kathy	IF	Branch Taken -> Check if PC jumps to target
Kathy	IF	Branch Not Taken -> Check if PC gets incremented
Kathy	IF	Roll Over -> Check if PC rolls over when crossing the Mem boundary

5.2. Instruction Decode

Aalap is assigned the responsiblity of through ahd through testing of the decode unit. The approach used here is to verify the accurate functioning of the intruction decode procedure.

Team Member	Design Unit	Test Name
Aalap	ID	Check if ZERO flag is asserted
Aalap	ID	Check if CARRY flag is asserted
Aalap	ID	Test the Operand selection
Aalap	ID	Check for invalid instructions in IDU
Aalap	ID	Verify the condition match logic

Aalap has discussed his initial plans here. But these plans are expected to change with more and more exploration of the architecture.

5.3. Instruction Execute

For each unit below would be performed a subset of tests.

Validate Signals

Team Member	Design Unit	Test Name
Niko	IE	Validate reg_source
Niko	IE	Validate reg_x_address
Niko	IE	Validate reg_y_address
Niko	IE	Validate reg_source
Niko	IE	Validate idu_operand
Niko	IE	Validate alu operation
Niko	IE	Validate Data Forward Control
Niko	IE	Validate reg_dst_address
Niko	IE	Validate Alu ADD
Niko	IE	Validate X propagation
Niko	IE	Validate Y propagation
Niko	IE	Validate out_port
Niko	IE	Validate port ID
Niko	IE	Validate Y propagation
Niko	IE	Validate out_port

Validate Instructions

For each instruction we will perform an assertion and unit test.

Team Member	Design Unit	Test Name
Niko	IE	Validate AND

Team Member	Design Unit	Test Name
Niko	IE	Validate ADDCY
Niko	IE	Validate STORE
Niko	IE	Validate SUB
Niko	IE	Validate SUBCY
Niko	IE	Validate XOR
Niko	IE	Validate OR
Niko	IE	Validate Call
Niko	IE	Validate XOR
Niko	IE	Validate OR

Validate Functionalities

Team Member	Design Unit	Test Name
Niko	IE	Validate RESET
Niko	IE	Validate COMPARE

5.4. Memory Access

There are a specific set of points that need to be considered while accessing the memory.

a. When the writeback stage expects them, memory addresses should be available.

b. The instructions where memory access is not the requirement should be directly forwarded.

Team Member	Design Unit	Test Name
Aalap	MA	Verify that the non-memory bound operations are forwarded with further processing
Aalap	MA	Check the boundaries of the scratchpad (64, 8-bit entries by default)

5.5. Write Back

Team Member	Design Unit	Test Name
Niko	WB	Validate carry
Niko	WB	Validate scratch pad
Niko	WB	Validate WB enabling

6. Assertions Testing

We are planning to use asserions in the form of a black box. For the most part we have two goals in mind:

1. Validate singal propagation is accurate
2. Validate Instructions are doing what they supposed to be doing

7. Test Plan Success Criteria

8. Citations

[1]. Mehta, Ashok B. SystemVerilog Assertions and Functional Coverage Guide to Language, Methodology and Applications. Springer International Publishing, 2016.