UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Diseño de un banco de pruebas para automatizar la verificación de microprocesadores RISC-V descritos en HDL mediante uso de metodología UVM.

Trabajo Fin de Grado

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

AUTOR/A: Barrera Comeche, Vicente

Tutor/a: Monzó Ferrer, José María

Cotutor/a: Gadea Gironés, Rafael

CURSO ACADÉMICO: 2021/2022

Resumen

RISC-V es una arquitectura para microprocesador libre que es muy usada en el ámbito académico y profesional, es enormemente flexive por lo que permite aplicar una variedad de diseños diferentes para implementar.

La metodología UVM (Metodología de Verificación Universal) se trata de un compendio de librerías para **SystemVerilog** que mediante una arquitectura bien definida por la metodología permite crear bancos de verificación muy potentes para cualquier diseño.

En este trabajo se implementará un banco de pruebas con UVM para verificar el RISC-V *singlecycle* que se diseña en el módulo de GITST "Integración de sistemas digitales", el objetivo final es que el banco se pueda reutilizar en el módulo para la comprobación de los diseños creados por los alumnos.

Las herramientas usadas para este proyecto son **SystemVerilog-VHDL-Assistant** para el diseño del banco de pruebas, **QuestaSim** para la simulación y comprobación y **Quartus** para la creación del *gold-model* que se usara para la verificación.

Resum

RISC-V és una arquitectura per a microprocessador lliure que es fa servir molt en l'àmbit acadèmic i professional, és enormement flexiu pel que permet aplicar una varietat de dissenys diferents per implementar.

La metodologia UVM (Metodologia de Verificació Universal) és un compendi de llibreries per a **SystemVerilog** que mitjançant una arquitectura ben definida per la metodologia permet crear bancs de verificació molt potents per a qualsevol disseny.

En aquest treball s'implementarà un banc de proves amb UVM per verificar el RISC-V *singlecycle* que es dissenya al mòdul de GITST "Integració de sistemes digitals", l'objectiu final és que el banc es pugui reutilitzar al mòdul per a la comprovació de els dissenys creats pels alumnes.

Les eines usades per a aquest projecte són **SystemVerilog-VHDL-Assistant** per al disseny del banc de proves, **QuestaSim** per a la simulació i comprovació i **Quartus** per a la creació del *gold-model* que s'usés per a la verificació.

Abstract

RISC-V is a free microprocessor architecture that is widely used in the academic and professional world, it is enormously flexible, allowing a variety of different designs to be implemented.

The UVM methodology (Universal Verification Methodology) is a compendium of libraries for **SystemVerilog** that, through a well-defined architecture for the methodology, allows the creation of very powerful verification banks for any design.

In this work, a test bench with UVM will be implemented to verify the RISC-V single-cycle that is designed in the GITST module "Integration of digital systems", the final objective is that the bench can be reused in the module for the verification of the designs created by the students.

The tools used for this project are **SystemVerilog-VHDL-Assistant** for the design of the test bench, **QuestaSim** for the simulation and verification, and Quartus for the creation of the gold-model that will be used for the verification.

Índice

Capítulo 1. Introducción

1.1 Objetivos

El objetivo de este trabajo es realizar un diseño de verificación en UVM para el RISC-V *singlecycle.* Se usará el banco como corrector de diseño por los alumnos, por lo que tiene que ser lo más automático posible, para que los estudiantes cambien lo mínimo.

1.1.1 Globales

Este objetivo se puede dividir en dos objetivos globales:

- entender el funcionamiento de UVM.
- aplicar las peculiaridades que necesite el RISC-V.

1.1.2 Parciales

Estos dos objetivos a su vez se dividen en diferentes subobjetivos:

Para lograr el objetivo de entender el funcionamiento de UVM,, se realizará un banco de pruebas sencillo. En este caso se hizo un banco de pruebas para un multiplicador *shift & add* multiciclo de dos operandos.

Para el objetivo de aplicar las peculiaridades que necesite el RISC-V (o del microprocesador, por si no quieres repetir RISC-V) se necesitará un *gold-model* que será el diseño de RISC-V *singlecycle* que realice y verifique con anterioridad en la asignatura de ISDIGI Después, se realizara un primer test a un DUT proporcionado, que es aparentemente estable, para verificar con el *gold-model* que sólo afecte a unas pocas instrucciones, para comprobar que la lógica implementada es correcta, por último, se aportara la repetición de algoritmos y estructuras para el resto de las instrucciones.

Un objetivo opcional que cumplir después de todo es el de crear una guía explicando cómo preparar y arrancar la simulación para que cualquier persona pueda usar el banco de pruebas.

1.2 Organización

1.2.1 Tareas

El cumplimiento de estos subobjetivos se organizará con diferentes tareas a realizar en el tiempo dedicado:

Para cumplir con el subobjetivo de banco de pruebas sencillo:

- Investigar el uso de la herramienta **SystemVerilog-VHDL-Assistant** y sus librerías con *templates* de UVM.
- Realizar la estructura básica de un banco de pruebas UVM.
- Añadir el multiplicador.
- Depurar el banco de pruebas.

Para cumplir con el subobjetivo del *gold-model*:

- Crear el banco de pruebas básico para el RISC-V.
- Revisión del diseño que será *gold-model*.
- Búsqueda de cómo crear el *gold-model* sin que sea visible desde el exterior.
- Implementar el *gold-model* en la estructura de UVM.

Para cumplir con el subobjetivo de las primeras verificaciones:

- Crear las estructuras para unas instrucciones, así como sus secuencias.
- Depurar estructuras y algoritmos.

Para cumplir con el subobjetivo de añadir todas las instrucciones:

- Duplicar y adaptar estructuras usadas en las primeras instrucciones.

- Depurar de nuevo con todo el conjunto además de controlar la cantidad de repeticiones para evitar que la simulación se alargue mucho.
- Revisión para posibles mejoras o peticiones.
- Implementar estas mejoras.
- Depuración de estas mejoras.
- Duplicación del proyecto con un RISC-V disfuncional para poner bajo estrés el banco.

1.2.2 Planificación de Tareas

Habiendo numerado todas las tareas, ahora hay que poner la duración de tiempo de cada una de ellas.

Actividades	Marzo				Abril				Mayo				Junio			
	Semanas				Semanas				Semanas				Semanas			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Investigar el uso de la herramienta																
Realizar la estructura básica																
Añadir el multiplicador																
Depurar banco de pruebas básico																
Crear el banco de pruebas																
Revisión del diseño *gold-model*																
Búsqueda para crear el *gold-model*																
Implementar el *gold-model*																
Crear las estructuras para instrucciones																
Depurar estructuras																
Duplicar y adaptar estructuras																
Depurar de nuevo																
Revisión para posibles mejoras																
Implementar mejoras																
Depuración de mejoras																
Duplicación del proyecto																

Tabla 1. Cronograma.

Capítulo 2. Conocimientos previos

2.1 Arquitecturas de microprocesador

Lo que se va a comprobar en este proyecto es el RISC-V, un microprocesador, por lo que es necesario conocer los diferentes partes que compone este tipo de dispositivos para tener algunas nociones sobre algunos términos.

2.1.1 ALU

Es la unidad aritmeticológica que puede realizar una gran cantidad de operaciones sencillas. El control es el que decide qué operación hará en cada momento y en algunas arquitecturas hay más de una ALU haciendo diferentes tareas.

2.1.2 Control

Es el que mediante indicadores gobierna el resto de los componentes, indicándoles que proceso deben efectuar en cada instrucción.

El flujo de funcionamiento es el siguiente:

Un registro llamado PC (Pointer control), hace de direccionamiento a una memoria que tiene las instrucciones que debe hacer en cada momento codificadas, después de un ciclo este registro cambia su valor, a la siguiente instrucción del programa. Cada arquitectura tiene unas instrucciones y una forma de codificarlas, el llamado código máquina. Al grupo de instrucciones de una arquitectura se le llama ISA (Instruction Set Architecture). El código máquina de la instrucción va hacia control, cuando lo interpreta el sistema controla todos los componentes, activando y desactivando diferentes indicadores y entregando parte de información de la instrucción necesaria para algunos componentes, de esta forma va indicando que debe hacer cada parte del microprocesador.

2.1.3 Registros

Es una pequeña memoria rápida que usa el micro para mover y procesar datos, llamada banco de registros.

Todos los cálculos se hacen siempre desde el registro y cada arquitectura usa diferentes cantidades y tamaños de registros. Es normal que los registros tengan asignados ciertos roles en diferentes arquitecturas, pero aun así se puede usar los registros libremente y a placer.

No todas las instrucciones usan únicamente los registros hay instrucciones que usan partes del código máquina, las llamadas inmediatas, e instrucciones que combinan registros con otros registros fuera del banco de registros.

2.2 RISC-V

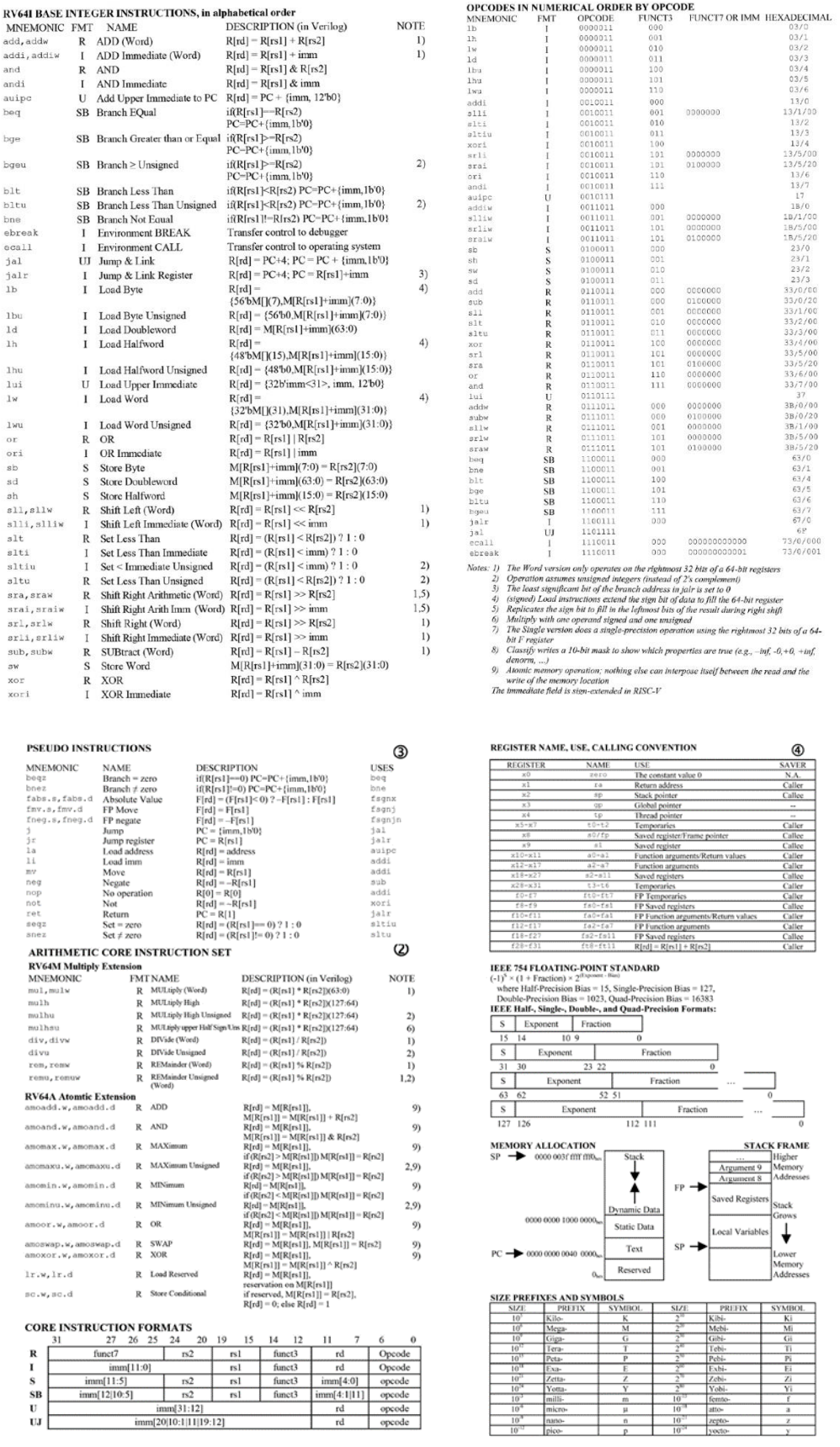
RISC-V es una arquitectura de ISA para microprocesadores. Es de código abierto por lo que ha desembocado en que una gran cantidad de empresas y universidades se interesen en desarrollarlo, creando un sinfín de módulos y extras que hace que la arquitectura siga creciendo y extendiéndose.

El tema es muy extenso así que el TFG se centrará en la versión de RISC-V que aquí se comprueba.

2.2.1 ISA

RISC-V tiene varios modelos de arquitectura de instrucciones como ya se mencionó, el que se usa en el módulo es RV32I, aunque en la Figura 1 se muestra el del RV64I, quitando las

instrucciones que solo son para más de 32 bits se obtiene con lo que correspondería al RV32I, además, en el módulo que se hizo no se usaron las menores de 32bits.



RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add,addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
addi,addiw	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2) PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
lb	I	Load Byte	R[rd] = {56'bM[](7),M[R[rs1]+imm](7:0)}	4)
lbu	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm](7:0)}	
ld	I	Load Doubleword	R[rd] = M[R[rs1]+imm](63:0)	
lh	I	Load Halfword	R[rd] = {48'bM[](15),M[R[rs1]+imm](15:0)}	4)
lhu	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm](15:0)}	
lui	U	Load Upper Immediate	R[rd] = {32b'imm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'bM[](31),M[R[rs1]+imm](31:0)}	4)
lwu	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sd	S	Store Doubleword	M[R[rs1]+imm](63:0) = R[rs2](63:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll,sllw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
slli,slliw	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra,sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai,sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl,srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli,srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
ld	I	0000011	011		03/3
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
lwu	I	0000011	110		03/6
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
addiw	I	0011011	000		1B/0
slliw	I	0011011	001	0000000	1B/1/00
srliw	I	0011011	101	0000000	1B/5/00
sraiw	I	0011011	101	0100000	1B/5/20
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
addw	R	0111011	000	0000000	3B/0/00
subw	R	0111011	000	0100000	3B/0/20
sllw	R	0111011	001	0000000	3B/1/00
srlw	R	0111011	101	0000000	3B/5/00
sraw	R	0111011	101	0100000	3B/5/20
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
2) Operation assumes unsigned integers (instead of 2's complement)
3) The least significant bit of the branch address in jalr is set to 0
4) (signed) Load instructions extend the sign bit of data to fill the 64-bit register
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift
6) Multiply with one operand signed and one unsigned
7) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
8) Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0,+0, +inf, denorm, ...)
9) Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location
The immediate field is sign-extended in RISC-V

PSEUDO INSTRUCTIONS ③

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = –F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = –R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

ARITHMETIC CORE INSTRUCTION SET ②

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul,mulw	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
mulh	R	MULtiply High	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhu	R	MULtiply High Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
div,divw	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem,remw	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
remu,remuw	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64A Atomtic Extension

amoadd.w,amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w,amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w,amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w,amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w,amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w,amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w,amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w,amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w,amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w,lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

REGISTER NAME, USE, CALLING CONVENTION ④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5–x7	t0–t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10–x11	a0–a1	Function arguments/Return values	Caller
x12–x17	a2–a7	Function arguments	Caller
x18–x27	s2–s11	Saved registers	Callee
x28–x31	t3–t6	Temporaries	Caller
f0–f7	ft0–ft7	FP Temporaries	Caller
f8–f9	fs0–fs1	FP Saved registers	Callee
f10–f11	fa0–fa1	FP Function arguments/Return values	Caller
f12–f17	fa2–fa7	FP Function arguments	Caller
f18–f27	fa2–fs11	FP Saved registers	Callee
f28–f31	ft8–ft11	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127,
Double-Precision Bias = 1023, Quad-Precision Bias = 16383

IEEE Half-, Single-, Double-, and Quad-Precision Formats:

	S	Exponent	Fraction
15	14	10 9	0
31	30	23 22	0
63	62	52 51	0
127	126	112 111	0

MEMORY ALLOCATION

STACK FRAME

SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi
10^{-3}	milli-	m	10^{-15}	femto-	f
10^{-6}	micro-	μ	10^{-18}	atto-	a
10^{-9}	nano-	n	10^{-21}	zepto-	z
10^{-12}	pico-	p	10^{-24}	yocto-	y

Figura 1.GreenCard

Las instrucciones pueden ser de distintos tipos, dependiendo de cómo se genera el código maquina en cada una serán de uno u otro tipo.

Las instrucciones que trabajan con dos registros se llaman R, estas están divididas en 6 bloques:

- El primer bloque, del bit 6 al bit 0, corresponde con el *opcode* este bloque es el único que tiene la misma asignación en todos los tipos de instrucciones.
- El segundo bloque, del bit 11 al bit 7, es el registro de destino, es decir el registro que recibirá el resultado de la operación, este bloque se repite en otros tipos.
- El tercer bloque, del bit 14 al bit 12, es el de función de 3 bits (fun3), el cual sirve definir exactamente qué operación va a realizar la ALU, este bloque también se repite.
- El cuarto bloque, del bit 19 al bit 15, es el primer registró de trabajo, este también se repite.
- El quinto bloque, del bit 24 al bit 20, es el segundo registro de trabajo, este es el que menos se repite.
- El sexto bloque, del bit 31 al bit 25, en este no se repite ninguno, este es la función de 7 bits (fun7), aunque solo cambia el penúltimo bit y el resto son 0 siempre, tiene la misma finalidad que fun3.

Después de haber explicado este tipo de formato el resto de los tipos es quitando bloques y cambiándolos por inmediatos, que son un valor escrito en la propia instrucción:

Las instrucciones que trabajan con un registro y un inmediato se llaman I, estas instrucciones eliminan el bloque quinto y sexto y lo cambia por un bloque de inmediato para operar. Tres instrucciones I son especiales ya que estas tienen fun7 al igual que R mientras que el quinto bloque sigue siendo un inmediato (slli srli srai). En las instrucciones de formato I también están las instrucciones de carga (lw) estas están aquí porque el formato es el mismo, pero no operan de la misma forma ya que usan la memoria de datos.

Las instrucciones de guardar datos son llamadas S (Store), estas instrucciones usan la memoria de datos para guardar información de los registros. Eliminan el sexto bloque y segundo bloque y lo cambian por un inmediato partido, donde cada parte del inmediato están en distintos bloques.

Las instrucciones de salto condicional son muy parecidas a las S, pero hay un cambio en los bits del inmediato creando las llamadas SB (Store Branch), el inmediato en este formato no solo está partido, sino que también está algo desordenado ya que un bit de la parte alta está en la baja por lo que requiere más trabajo para montar el inmediato.

Las instrucciones que sirven para añadir datos en la parte alta de un registro se llaman U (Upper), estas tienen inmediatos muy largos usando los bloques del tercero al sexto. Hay una instrucción que puede guardar en registros la suma del PC con el inmediato como bits más significativos del PC (auipc). Se usa junto con una instrucción de salto para moverse rápidamente grandes distancias en la memoria de instrucciones, (jalr), esto puede servir si la memoria de instrucciones esta compartida con la de datos y se necesita de un gran desplazamiento para empezar el programa.

Las instrucciones de saltos no condicionales son muy parecidas al formato U, la diferencia está en que, al igual que las de SB, el inmediato esta desorganizado, hay bits de la parte alta en la baja y viceversa. Estas instrucciones son llamadas UJ (Upper Jump), este tipo de instrucciones guardan la posición del PC en los registros antes de ejecutarse, cosa que las de salto condicional no. Una de las instrucciones de salto no condicional tiene un formato I en vez de UJ (jalr).

Estos son todos los formatos en que se codifican las instrucciones, es necesario saber todo esto para la creación aleatoria de instrucciones que se usara en la verificación.

2.2.2 Arquitectura del módulo de RISC-V

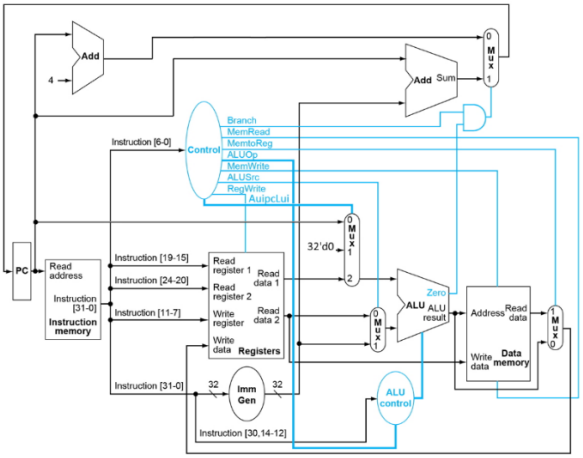


Figura 2.Esquema de RISC-V

En la Figura 2 se puede ver el esquemático, los módulos y las memorias, objeto de la verificación.Para la verificación se extraen las memorias y se simulan sus valores, todas las entradas y salidas de las memorias son parte de las variables de verificación, además para facilitar la verificación se usan las entradas del registro de dato de escritura y de habilitación de escritura.

En la memoria de datos también hay habilitación de lectura, pero no todo el mundo la usa o la coloca. Hay quien la habilitación de lectura es el valor contrario de la habilitación de escritura, o quien directamente no implementa habilitación de lectura y la memoria se lee todo el rato. El banco de pruebas ha de poder verificar sin tener en cuenta esto. La mejor forma es ignorar dicha habilitación, lo cual podría suponer un *Hazard* teniendo en cuenta que al no haber memoria creamos nosotros la variable, por lo que, en caso de haber un problema en esa habilitación no se vería. Sin embargo, solo hay una instrucción que use dicha habilitación y es muy difícil que aparezca un error y aunque apareciera se acabaría saliendo en otra etapa de verificación. Por si acaso se debe informar sobre este Hazard a los alumnos para evitar inconvenientes.

Hay que tener en cuenta algunas cosas en el diseño para que la verificación sea correcta, esto normalmente esto no sería así, pero como en este caso la verificación esta creada antes que el diseño, hay que sacrificar un poco de libertad de diseño en pro de la adaptabilidad:

- El tamaño del direccionamiento depende del tamaño de las memorias y para evitar gastar todo el espacio del chip ese valor se parametriza, sin embargo, el direccionamiento de la verificación está pensado con su máxima capacidad que es de 32 bits, es normal que en diseños de alumnos el PC tenga menos de 32 bits, por lo que para evitar que los alumnos toquen parámetros de la verificación se obligara que los direccionamientos de memoria sean de 32 bits.
- El PC debe cambiar en cada flanco de subida de reloj sin tiempo de espera para que la verificación salga bien.
- El direccionamiento de las memorias es de Word a Word sin embargo el direccionamiento del módulo es de byte a byte, es porque a la hora de introducir el direccionamiento a la memoria se saltan los dos primeros bits. Para verificar eso no es así por lo que es importante que en el diseño del módulo el salto de los bits ocurra en el top junto con las memorias y no en el módulo, ya que el módulo que es lo que se va a comprobar no debe tener ese salto.

2.3 UVM (Universal Verifcation Methodology)

UVM es un conjunto de librerías y metodologías estandarizadas para el diseño de verificadores para IPs de diseño en lenguajes de descripción. Su antecesor fue OVM, y la desarrolló **Acellera**, un aglomerado de empresas para estandarizar en la industria sistemas de automatizar los diseños en la electrónica.

Figura 3.Ficha de presentación de Accellera

Las librerías son un conjunto de clases de **SystemVerilog** de código abierto, que con ayuda de macros arregladas también de UVM, se puede crear un conjunto de componentes para la verificación con un mismo patrón para una mejor reutilización, además hace más fácil la utilización de metodologías de cobertura funcional y generación de valores para las entradas con restricciones para diferentes casos. Otras de sus características pueden ser:

- Puede verificar IPs diseñadas con lenguajes diferentes como **Verilog**, **VHDL** o **SystemC**.
- Como lo desarrollo **Accellera**, todas las empresas que participan con **Accellera** soportan y utilizan este estándar, al ser empresas con cierta influencia provoca una mayor utilización fuera del conjunto de **Accellera**, empresas como **Siemens**, **Intel**, etc.
- Utiliza un esquema en árbol reutilizando clases extendidas, heredando funciones y utilidades a partir de una clase única.
- Con las macros que contiene la librería se puede automatizar muchas de las funcionalidades que necesitan de una intervención por parte del usuario.
- Los diferentes componentes se pueden conectar entre ellos con puertos de interfaz (TLM), que permite aislar los componentes unos de otros, para facilitar la extensión y reutilización de distintos componentes.
- Puede encargarse de la verificación funcional de grandes diseños para diferente hardware, grandes circuitos integrados hechos desde diseños en bloques, hasta diseños a nivel de semiconductor.
- La librería tiene una gran cantidad de funciones, clases y macros, pero el usuario que está diseñando la verificación solo hace uso de una pequeña parte, el resto es usado por la propia librería de forma interna.

2.3.1 Estructura de un testbeanch en UVM

A partir de esta sección hasta el final del capítulo se hablará de la estructura que componen un banco de pruebas hecho con UVM, así como todos los componentes que se usan para ello. Esta información es la que se aplicó para realizar el *testbeanch* del RISC-V.

2.3.2 Jerarquía UVM

Todas las clases de UVM están dentro de una red de jerarquías de clases que extienden unas de otras y se expanden a clases del usuario. Las clases con extensiones a otras clases tienen acceso a las funciones y *tasks* junto con sus señales y valores de la clase padre y del resto de las extensiones. Gracias a esto las ultimas clases como *uvm_test* y *uvm_sequence* tienen acceso a los diferentes elementos de la clase *uvm_object*. También las clases que extiendan *uvm_test* o *uvm_sequence* tendrán acceso a sus elementos y a los de *uvm_object*.

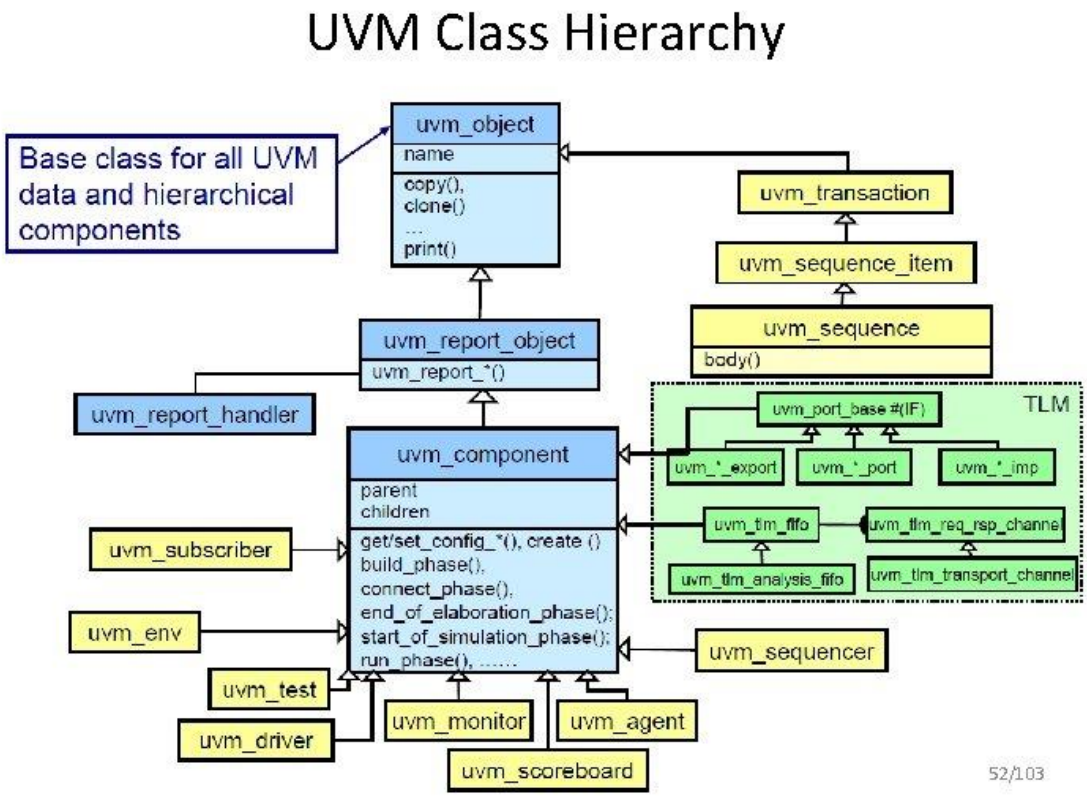


Figura 4.Jerarquía de clases en UVM

2.3.3 Clases

Las clases más significativas son *uvm_object*, *uvm_transaction* y *uvm_component* son las clases que más métodos y datos añaden, y ahora se explicaran un poco más cada una:

- La clase base para todas las demás es *uvm_object*, de esta se expanden el resto de las jerarquías. En ella se definen las funcionalidades más básicas que usan todas las demás clases, cosas como *copy*, *print*, *create*, *compare*…
- La clase que expande a la mayoría de los componentes es *uvm_component*, las clases principales que se usan para crear los componentes de UVM (*uvm_agent*, *uvm_env*, *uvm_driver*…), esta clase hereda también de *uvm_report_object* y esta a su vez de *uvm_object*. Esta clase contiene estas funcionalidades.
    - Contiene una serie de funciones y *task* para moverse y buscar diferentes componentes a través de la jerarquía.
    - Mantiene un flujo de actuación que los componentes realizan durante la simulación. Las diferentes secciones retienen la simulación mediante *callbacks* para que se ejecuten en el orden correcto(*build¬>connect¬>run*…).
    - Mediante métodos se puede crear y cambiar las configuraciones de la topología y de algunos parámetros de los diferentes componentes durante la fase de *build* como en fases de más adelante
    - Mediante unas macros se puede usar unos métodos para controlar los mensajes del *transcript*, incluido errores, advertencias y errores fatales.
    - Tiene unas funciones y *task* que registran los movimientos de paquetes en las transacciones ocurridas en el componente, todo esto se almacena en una base de datos que contiene los componentes y las cantidades de transacciones registradas.
    - Para poder usar *uvm_factory* y crear nuevos objetos y componentes, pueden usar una interfaz de creación.
- Para realizar las transacciones entre componentes se usa la clase *uvm_transaccion* como base para las clases de transacción. Hereda de *uvm_object* por lo que es la misma base de

trabajo que *uvm_component*. Esta clase controla el tiempo de los paquetes mediante una interfaz que contiene: ajustes para marcas de tiempo, notificaciones eventuales y es el que contiene el registro de las transacciones de los componentes. Esta clase no la usa el usuario, el usuario usa la siguiente clase de la jerarquía para crear los datos y métodos que contiene los paquetes y la siguiente para crear las secuencias de *test* para enviar paquetes al *uvm_sequencer*.

-

2.3.4 Macros

Una macro sustituye una cantidad de código por una *tag* que genera el código a la hora de compilar, por lo que no es como las funciones o *tasks* y no realiza llamadas. A una macro también se le puede poner argumentos para modificar el código generado.

```
`define tag(argumento1, argumento2,...) código;
```

A partir del pre-compilado del lenguaje de **SystemVerilog** se va generando el código en función de la *tag* y los argumentos que después se compilaran.

Con *`include uvm_macros.svh* se incluyen todas las macros que el usuario puede necesitar durante la creación de los componentes de UVM. En este fichero están organizadas todas macros de diferentes archivos y demás como un paquete de macros.

```
`include "uvm_macros.svh"
```

Las macros que más se utilizan son:

- Para registrar un tipo de clase en la fábrica que extiende de alguna clase que hereda de *uvm_component* se usa *uvm_component_utils*.
- Si la clase que se registra en la fábrica hereda de *uvm_object* en vez de *uvm_component* se usa *uvm_object_utils*.
- Para registrar variables en la fábrica y añadir les funcionalidades se usa *uvm_field_int*, esta añade métodos como *copy()*, *compare()* y *print()*, para usar esta macro es necesario que este dentro de un *begin* y *end* de macros, con *uvm_*_utils_begin\end*.
- Las macros de notificaciones sirven para crear texto en el *transcript* además de controla los resultados de la simulación, cada vez que se usa una de estas macros se registran y al final de la simulación en un resumen pone la cantidad de veces que una notificación con la misma etiqueta ha sido utilizada, si se usa *uvm_info* se debe indicar el nivel de importancia para saber dependiendo de que filtro de verbosidad este activo debe imprimirse o no, luego están *uvm_warning* y *uvm_error* que tienen una importancia especial reservada para el resumen final, por ultimo esta *uvm_fatal* que además de notificar detiene la simulación.

Podemos ver en el siguiente trozo de código como se genera un componente extendido de *uvm_monitor* que hereda de la clase *uvm_component* utilizando la macro para registrarla en la fábrica y el constructor para instanciar la clase.

```
class agente_monitor extends uvm_monitor;

  `uvm_component_utils(agente_monitor)


  function new (string name = "agente_monitor", uvm_component parent = null);
    super.new(name,parent);
  endfunction: new

enclass
```

2.3.5 Fases

Como ya se ha mencionado antes UVM trabaja a través de fases que van atrapando la simulación para mantener un orden claro y conciso. Esto asegura que todos los componentes que constituyen

la simulación estén correctamente generados y conectados. Gracias a esto todos los componentes funcionan de forma sincronizada y ordenada.

Las fases usan mecanismos de *callback* para controlar cual está funcionando en cada momento. La clase *uvm_component* ya tiene unas cuantas fases para todos los componentes que sus respectivos *callback*. Así todas las clases que deriven de *uvm_component* tiene las mismas fases y los mismos *callback*, de esta forma todos los componentes se generan en el mismo orden.

He aquí el orden de ejecución:

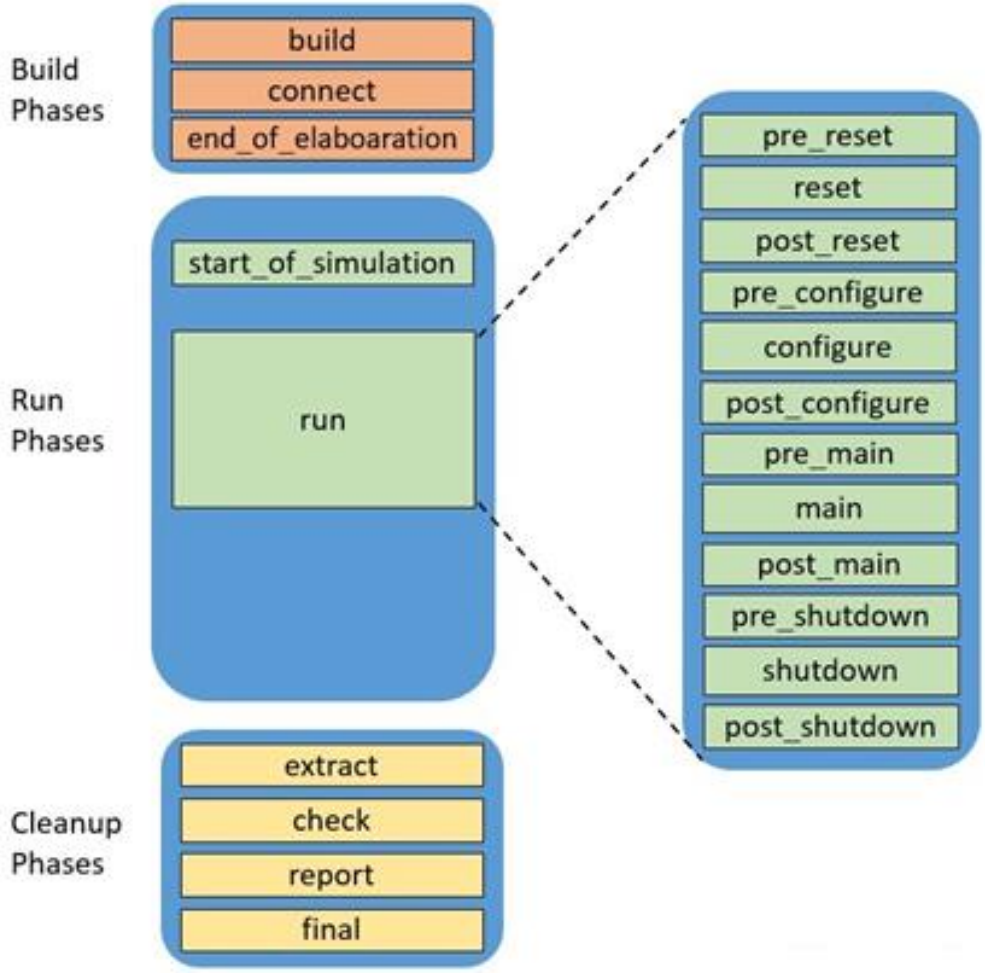


Figura 5.Flujo de ejecución de las fases

Todas las fases pueden ser agrupadas en tres categorías:

1. **Build phases:** Son las fases de construcción y conexión, estas fases comienzan antes de que arranque la simulación por lo que todas se ejecutan en el segundo 0.
2. **Run phases:** Es la fase de simulación *run_phase*, esta fase está en bucle hasta que se cumplen los requisitos de la simulación, durante esta fase el tiempo de simulación está corriendo y esta fase se puede dividir en sub-fases.
3. **Clean_Up phases:** Con estas fases cierran y recogen los resultados haciendo un resumen final de todas las notificaciones durante la simulación.

Mediante esta tabla se puede ver el orden y el método que usa cada fase además de para que sirven:

Categoría	Nombre	Tipo de método	Descripción
Build	build_phase	function	Crea los componentes con la fábrica.
	connect_phase	function	Conecta todos los puertos TLM de los componentes como se han configurado.
	end_of_elaboration_phase	function	Configura las ultimas variables de los componentes, para ajustar todo antes de que comience la simulación.
Run	start_of_simulation_phase	function	Esta fase no hace correr la simulación por lo que depende de quien no la considera una **Run phase**, pero esta fase solo notifica por lo que a tiempo de programa no se puede considerar.
	run_phase	task	En esta fase se realizan las actividades principales de la simulación como generar datos transportara paquetes, capturar señales y comprobarlas. Es la única que se implementa como *task* y que consume tiempo de simulación. Las *run_phase* de todos los componentes funcionan a la vez, aunque algunos componentes pueden frenan las *run_phase* de otros, mediante métodos.
Clean	extract_phase	function	Recoge todas las notificaciones informativas que hayan podido salir durante la simulación del *coverage* y *scoreboard*
	check_phase	function	Similar que la de arriba, pero esta solo recoge las notificaciones de *warning*, *error* o *fatal*.
	report_phase	function	Es la fase que crea el resumen en el *transcript* o en un archivo si está configurado, con lo que han recogido las otras dos fases.
	final_phase	function	Termina todas las tareas pendientes del *testbeanch* y finaliza definitivamente la simulación.

Tabla 2. Resumen fases

2.3.6 Objeciones

El control de las fases se realiza mediante objeciones que intercambian entre las jerarquías de los componentes. Todas las fases tienen sus propias objeciones que le permiten a los componentes saber si los otros han finalizado su fase para saber cuándo terminar con la fase y comenzar con la siguiente de esta forma los componentes aseguran su sincronización.

Al comienzo de cada fase todos los componentes levantan una objeción de fase con *raise_objection*, mientras la objeción esté activa todos los componentes realizan las actividades propias de la fase, para poder salir de la fase todos los componentes deben bajar la objeción con el uso de *drop_objection*.

Hay un registro con todas las objeciones que los componentes han levantado para tener controlado cuantas objeciones están activas, conforme los componentes van bajando las objeciones el registro va reducción la cantidad objeciones activas. Cuando la cantidad de objeciones activas llegan a cero la fase se considera terminada.

2.3.7 Factoría

Para tener registrado y controlados todos los objetos que se crean en UVM se usa la fábrica como tabla de búsqueda mediante registros. Para utilizar la fábrica requiere un orden:

1. Primero se debe registrar el componente en la fábrica para tener acceso a él desde la tabla, esto se puede realizar mediante las macros anteriormente mencionadas, *`uvm_object_utils* y *`uvm_component_utils*.
2. Los constructores de las clases deben ser siempre iguales con los mismos argumentos dependiendo de si es un *component* o *object*, esto se debe a que los constructores son virtuales.
3. Lo último es crear el componente o el objeto mediante el método *name_component::type_id::create()*. Este método se realiza en la fase *build_phase*.

Con un ejemplo se puede ver cómo crear un componente extendido de *uvm_monitor* con la fábrica, los anteriores pasos fueron mencionas en el fragmento de código anterior:

```
                    […]
virtual function void build_phase(uvm_phase phase);
agente_monitor m_monitor = m_monitor::type_id::create ("m_monitor", this);
//Construcción del componente
```

2.3.8 Conexiones TLM

El TLM es el sistema de modelado en nivel de transacción para separar la implementación de la comunicación entre componentes de las partes funcionales de los distintos elementos. De esta forma podemos aislar los componentes entre si para una mayor flexibilidad, permitiendo actualizar las distintas partes sin que la comunicación conforme un problema. Con esta metodología de comunicación a nivel de transacción permite la comunicación entre todos los componentes que usen la misma interfaz de TLM.

UVM TLM es la biblioteca con todas las interfaces a nivel de transacción que contiene UVM, *export*, *ports*, *analysis_**, etc. La interfaz TLM más usada es TLM1 ya que esta proporciona transacciones con control de actuación, gracias a esto las transacciones pueden ser bloqueantes, hasta que no termine la transacción no continua el resto, o no bloqueantes, que permite hacer varias transacciones a la vez.

Figura 6.UVM TLM1

2.3.9 Interfaz TLM

Para usar la interfaz TLM tiene declarada algunas tareas para recibir y enviar las transacciones. Hay 4 tipos diferentes de métodos que se pueden usar:

- **Blocking:**
  - *<port>.put(trans):* pone un paquete de transacción en el puerto y espera a que alguien lo reciba.
  - *<port>.get(trans):* recoge un paquete del puerto o espera a tener uno para recibirlo y consume el paquete del puerto.
  - *<port>.peek(trans):* hace lo mismo que *get* salvo que no consume el paquete de transacción.

- **Non-Blocking:**
  - *<port>.try_put(trans):* pone un paquete de transacción en el puerto si es posible y si hay alguien para recibirlo, sin bloquear el componente. Si alguien recibe el paquete el método devuelve un 1, si no es capaz de enviar el paquete devuelve un 0.
  - *<port>.can_put():* comprueba el estado del puerto para saber si se puede o hay alguien para recibir un paquete de transacción, no requiere de ningún argumento, de esta forma podemos saber si *try_put* tendrá éxito sin usar el método.
  - *<port>.try_get(trans):* permite recibir un paquete de transacción del puerto si hay una disponible. Si recibió una transacción devuelve un 1, si no recibe nada devuelve 0.
  - *<port>.can_get():* comprueba si hay algún paquete listo para ser recibido, pero no hace falta un argumento, devuelve un 1 o un 0 si hay un paquete o no. Es similar que *can_put* pero para la recepción.
- **Transport**:
  - *<port>.transport(req, rsp):* realiza un intercambio de paquetes de transacción, es decir recibe y envía un paquete, bloquea el desarrollo hasta que alguien consuma el paquete.
  - *<port>.nb_transport(req, rsp):* hace lo mismo que *transport* pero esta no es bloqueante, por lo que si no hay un paquete o nadie recibe su paquete termina y devuelve un 0, si la transacción se realiza con éxito devolverá un 1.
- **Analysis**:
  - *<port>.write(trans):* hace lo mismo que el método *put*, pero el paquete que pone puede ser consumido por más de un componente permitiendo a varios componentes recibir el mismo paquete, además a diferencia del otro método este no bloquea simplemente deja el paquete puesto en el puerto y continua con la ejecución.

2.3.10 Comunicación entre puertos TLM.

Hay tres tipos de puertos básicos de los que derivan todos los demás, cada uno de estos puertos se pueden conectar de manera diferente:

- Los puertos *port* sirven para enviar transacciones a puertos disponibles, se pueden conectar a el resto de los puertos y puede funcionar como solo envió o de recibo y respuesta.

```
uvm_*_port #(T)       //unidirectional – T type of the transaction
uvm_*_port #(REQ, RSP) //bidirectional – REQ type request RSP type response
```

- El puerto *export* sirve para recibir transacciones y reenviarlas a otros componentes que estén a un nivel por debajo de abstracción. Se puede conectar a otros puertos *export* y a imp *ports* y también se puede configurar como recepción y respuesta.

```
uvm_*_export #(T)        //unidirectional - T type of the transaction
uvm_*_export #(REQ, RSP)   //bidirectional - REQ type request / RSP type response
```

- El ultimo puerto es el imp *port*, este puerto es para recibir respuestas y suele ser la última etapa de envió de transacción, este puerto no se conecta a ninguno más y también se puede configurar como envió y respuesta.

```
uvm_*_imp #(T, IMP) //unidirectional
uvm_*_imp #(REQ, RSP, IMP, REQ_IMP, RSP_IMP) //bidirectional
```

La interfaz que se crea con los puertos es del tipo productor y consumidor, el productor tiene puertos *port* y genera los paquetes, y el consumidor tiene puertos imp *port*, si el consumidor está dentro de otros componentes estos conectan el productor con el consumidor a través de puertos *export*. En el caso de que sea el productor el que está dentro de otros componentes todos estos usan puertos *port*.

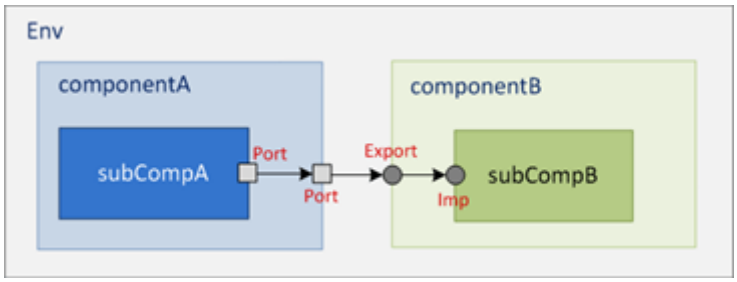


Figura 7.Comunicación TLM básica.

Teniendo en cuenta lo dicho las conexiones que se pueden realizar son:

- *port-to-port*
- *port-to-export*
- *port-to-imp*
- *export-to-export*
- *export-to-imp*

2.3.11 TLM FIFO y TLM Analysis FIFO

uvm_tlm_fifo#(T) es un componente intermediario en las transacciones que sirve para acumular paquetes. Con este componente el productor puede seguir enviado paquetes sin tener que bloquearse siempre cuando haya espacio en la FIFO, mientras el componente consumidor puede ir extrayendo los paquetes de la fifo sin bloquearse siempre que haya un paquete en la fifo.

Para introducir los paquetes en la fifo se debe usar el *put_export* y para extraer los paquetes se debe usar *get_export*, además si se quiere extraer el paquete sin consumirlo se puede usar *get_peek_export*. También se pueden usar los métodos normales de puertos.

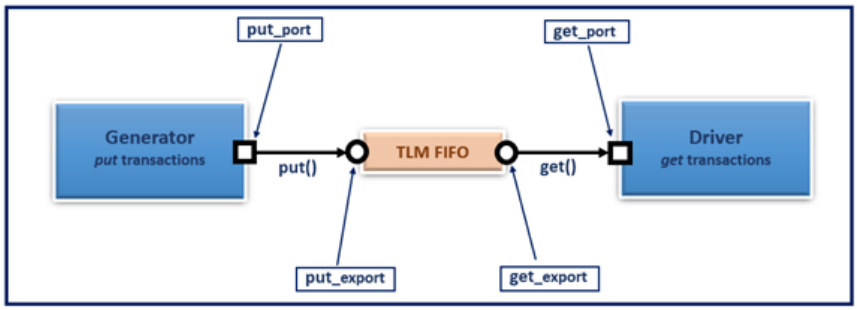


Figura 8.TLM FIFO entre productor y consumidor.

uvm_tlm_analysis_fifo#(T) deriva de *uvm_tlm_fifo#(T)*. A diferencia de la otra fifo esta tiene un tamaño ilimitado, y un puerto de tipo *analysis_export* que le permite usar el método *write()*. Se usa en donde se vaya a usar un puerto del tipo *uvm_analysis_imp* como búfer entre una interfaz

de transacción de tipo *analysis*, de esta forma si hay varios generadores (como monitores) se puede integrar todo con un solo puerto para el consumidor.

2.3.12 TLM Analysis Ports

Para usar los métodos de transacción TLM no bloqueantes se necesita usar de los puertos *uvm_analysis_port*. A estos puertos se pueden conectar múltiples componentes para crear una red de puertos donde todos los generados conectados envían a todos los consumidores. Este tipo de puerto se puede crear y no conectar a ninguna interfaz y no pasaría nada a diferencia de los puertos normales que podrían bloquear la simulación. Se suelen poner en aquellos componentes que necesiten enviar a muchos otros como un monitor que necesite enviar a *covergroups* y *scoreboards*.

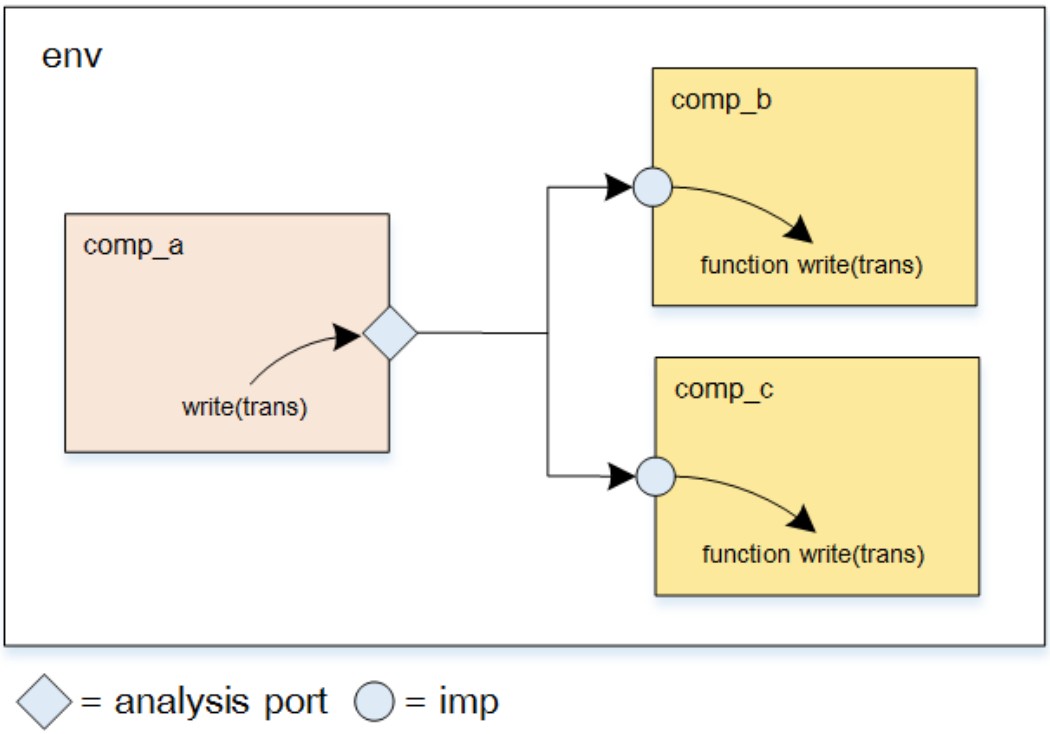


Figura 9.Conexiones puertos de análisis.

Hay los mismos puertos que los normales, pero *analysis port*:

- *uvm_analysis_port* enviara un paquete de transacción a todos los consumidores de la red que estén conectados con los otros puertos.
- *uvm_analysis_imp* recibirá un paquete de algún *analysis_port*, normalmente un *analysis_imp* se conecta a un solo *analysis_port*.
- *uvm_analysis_export* traspasa un paquete de *analysis_port* a un puerto de *analysis_imp* o *export* que este dentro del componente.

Para enviar los paquetes de transacción por los puertos *analysis_port* solo se usa un método, *write()*. Esta es la única función que envía el paquete a la interfaz para que todos los consumidores reciban el paquete. Para que todo funcione el componente consumidor debe tener la función *write()* implementada. Por suerte UVM tiene una clase llamada *uvm_subscriber* de la cual los *scoreboard* pueden extender que ya tiene la función *write()* implementada a través de un *uvm_analysis_imp* instanciado como *analysis_export*.

```
virtual class uvm_subscriber #(type T=int) extends uvm_component;
  typedef uvm_subscriber #(T) this_type;

  uvm_analysis_imp #(T, this_type) analysis_export;

  function new (string name, uvm_component parent);
```

```
    super.new(name, parent);
    analysis_export = new("analysis_imp", this);
  endfunction

  pure virtual function void write(T t);

endclass
```

2.3.13 Base de datos de configuración

La clase de configuración proporciona un registro centralizado de la información de distintos componentes. Toda la información puede ser leída y modificada por todos los componentes que tengan acceso a la base de datos. Toda la información se organiza por nombre y por tipo y se puede buscar la información a través de las dos tablas.

La clase que proporciona todo esto se llama *uvm_config_db*, la cual solo usa dos métodos:

- Para almacenar o modificar información se usa *set()*.

```
void uvm_config_db#(type T = int)::set(uvm_component cntxt, string inst_name, string
field_name, T value);
```

  - El argumento T es el tipo de la información que se quiere guardar o modificar. Los tipos de elementos son escalares, control de clases, colas, listas o interfaces virtuales.
  - *cntxt*, *inst_name* se usan para decir que componentes tienen acceso a la información original para indicar la ruta jerárquica que acceden los componentes desde la base de datos.
  - *field_name* es el nombre que se guardara en la tabla de nombres para buscar en la base de datos a través del nombre.
  - *value* es el objeto que se guarda en la base de datos, puede ser un valor o una instancia.

- Para recuperar la información se usa el método *get()*.
  - Los argumentos son muy similares al anterior método solo que esta vez en vez de usarlos para guardar la información los usara para buscar la información.
  - El único argumento que cambia es valor ya que este no se usa para buscar la información, si no que será la variable donde se guarde el valor recuperado de la base de datos.

Se puede ver un ejemplo de configuración muy usado para poner una interfaz virtual en la base de datos, donde *multi_if* es el tipo del elemento y *vif* es la interfaz a conectar, en el caso del argumento *inst_name* el "*" indica que cualquier elemento tiene acceso a la interfaz virtual original.

```
multi_if vif(clk,reset);  //interface instance

uvm_config_db#(virtual multi_if)::set(null,"*","multi_if_vif",vif);  //set method
```

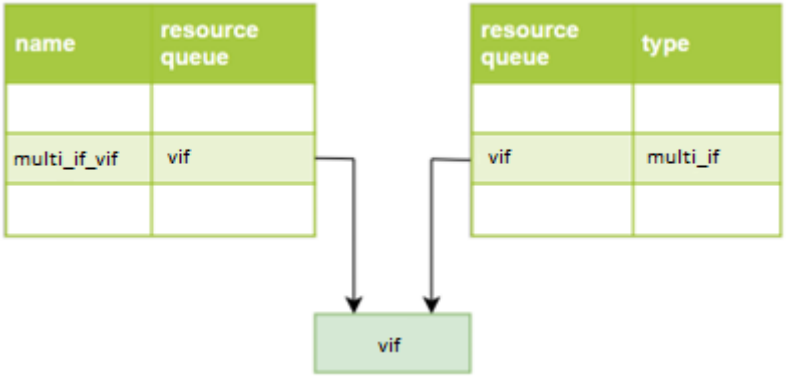


Figura 10. Tabla de la base de datos de configuración.

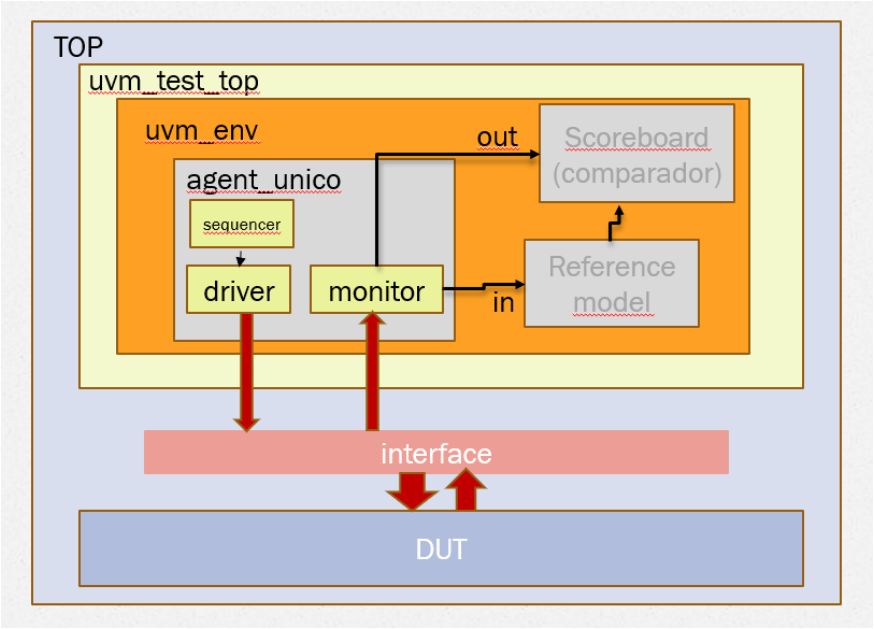


Figura 11. Estructura del banco de pruebas

2.3.14 TOP

El TOP de la simulación es donde comienza la compilación, por lo que debe tener *includes* con los paquetes de compilación de los componentes, también es donde se instancian los elementos que no forman parte del *environment* de UVM como el DUT o la interfaz que se usa en la comunicación entre el DUT y los agentes.

Además de todo lo dicho anteriormente en el top también esta:

- *run_test()* es el método global que inicia la simulación desde el test en la factoría para empezar las ejecuciones de *run_phase*. Cuando se ejecuta la tarea *run_test()* esta se ejecuta desde una instancia de *uvm_root*, desde donde comienza la factoría a comprobar los registros y configuraciones básicas para asegurarse que el nombre de test existe y se puede arrancar su *run_phase*. En la *run_test* se ejecuta el método *uvm_run_phase* el cual comienza la cadena de fases del resto de componentes. Después de terminar la simulación el **Report Server** hace un resumen de todas las notificaciones generadas por los componentes, los *info*, *warning* y *error* que ha ido registrando. Al final con *report_summarize()* muestra el resumen en el *transcript* y finaliza la simulación con *$finish*.

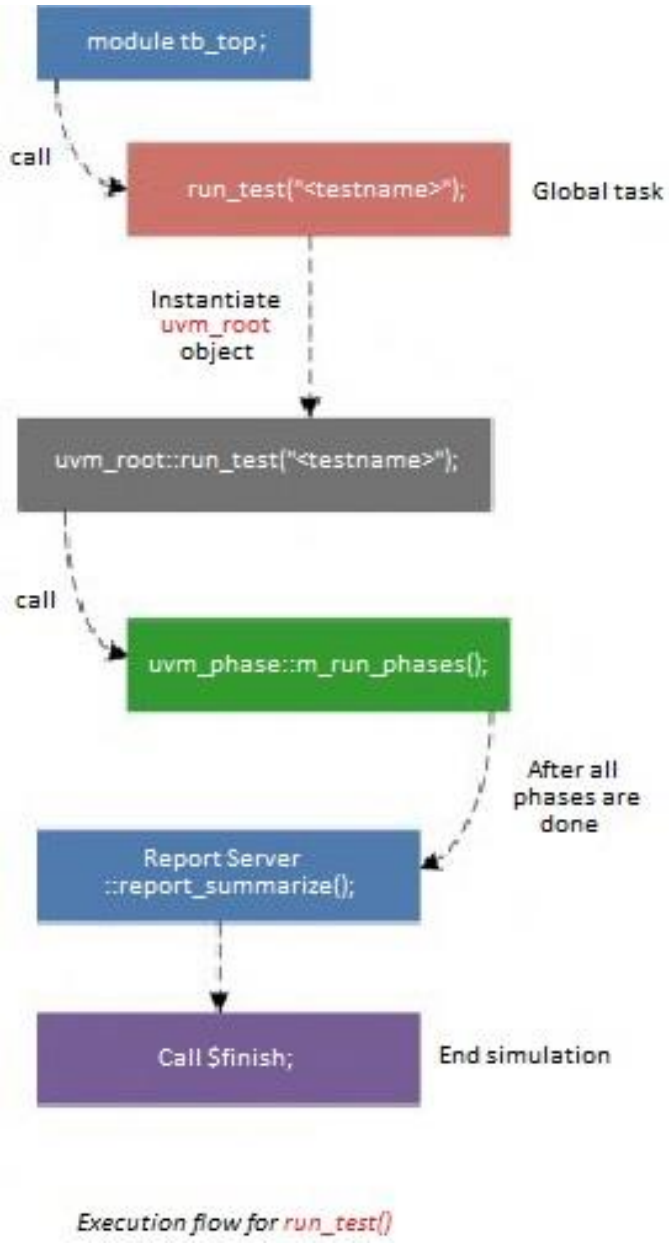


Figura 12. Flujo de ejecución de run_test()

- Se crean las interfaces virtuales y se guardan en la base de datos de *uvm_config_db* usando la estructura vista del método *set()*. Estas interfaces se usarán por los componentes usando el método *get()* de *uvm_config_db*.
- Se crean las señales de reloj mediante *always* o con algún modulo, también se crea una señal de *reset* inicial para el DUT de la misma forma.

2.3.15 Test

El *test* o *tests*, ya que se puede realizar la simulación con un único test o si el módulo es muy complejo se pueden hacer varios test para comprobar los casos por separado. Para ello se crea un banco de prueba para el módulo y los *test* se independizan del propio banco de pruebas permitiendo crear muchos test con solo un banco. Al cambiar el *test* el entorno se configura y se prepara para los distintos casos de test permitiendo una flexibilidad inmediata. Estos *tests* pueden cambiar las configuraciones del *environment*, activar o desactivar componentes, cambiar la base de datos de *uvm_config_db* y cambiar la secuencia del secuenciador que va a usar el agente.

Para crear una clase de *test* tiene que extender de *uvm_test* que esta deriva de *uvm_component*. Para elegir un *test* hay dos formas de hacerlo:

1. Usando un argumento en el método *run_test* con una etiqueta con el nombre del *test* guardado en la fábrica.

```
run_test ("model_test");
```

2. Indicando al arrancar la simulación en una línea de comando el nombre del *test* en la variable UVM_TESTNAME.

```
<SIMULATION_COMMANDS> + UVM_TESTNAME = model_test
```

2.3.16 Entorno

El entorno es donde se engloba toda la arquitectura de UVM, es donde se realizan todas las conexiones de los agentes con distintos componentes y las configuraciones según el DUT que se esté comprobando. Un *environment* típico seria uno con un *scoreboard*, un agente activo para controlar la interfaz del DUT y un modelo de referencia que sirve para comparar en el *scoreboard*. Para crear este componente debe extender de *uvm_env* que deriva de *uvm_component*.

2.3.17 Agentes

Un agente puede contener un monitor, un *driver* y un *sequencer* instanciados y conectados con puertos de interfaces de TLM. Un agente puede ser pasivo o activo dependiendo de que componentes contenga . Para crear este componente la clase debe extender de *uvm_agent* y esta deriva de *uvm_component*.

- El agente activo es aquel que contiene un componente de cada: un monitor, *driver* y *sequencer*. Este agente puede introducir y leer señales de DUT a través de la interfaz y enviarlas a *scoreboard* y cobertura funcional.
- En cambio, el agente pasivo solo contiene un monitor y solo puede leer señales de una interfaz, por lo que solo puede leer datos del DUT o de algún módulo de referencia.

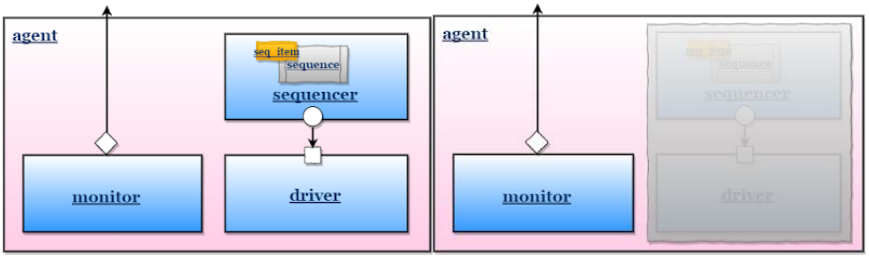


Figura 13. Tipos de agente

Un agente es activo o pasivo dependiendo de los componentes. También se puede cambiar en la base datos para activar o desactivar los componentes para cambiarlos entre activos o pasivos. Si no se hace nada todos los agentes se consideran activos.

```
uvm_config_db #(int)::set(this, "path_to_agent", "name_agent", UVM_PASSIVE);
uvm_config_db #(int)::set(this, "path_to_agent", "name_agent", UVM_ACTIVE);
```

2.3.18 Monitor

El componente que se encarda de registrar las señales de la interfaz que se deseen y meterlas en un paquete de transacción es el monitor, este elemento es pasivo y solo actúa cuando algo se lo indica. Una vez creado el paquete se envía a componentes consumidores por puertos de nivel de transacción, como suelen haber varios componentes consumidores suelen ser puertos *analysis*. Estos datos se envían a *scoreboard* para su comparación y análisis o a coberturas funcionales. Para crear este componente se tiene que extender de *uvm_monitor* que deriva de *uvm_component*.

2.3.19 Driver

El componente que se encarga de escribir en la interfaz del DUT es el *driver*. Cuando recibe un paquete de transacción del *sequencer* lo introduce al DUT a través de la interfaz. Una vez enviado los datos a las señales envía una respuesta al *sequencer* con un paquete de transacción. Para crear este componente se debe extender de *uvm_driver* que deriva de *uvm_component*.

Los paquetes de transacción que se envían entre el *sequencer* y el drive se hacen mediante unos puertos TLM especiales, *seq_item_port* y *seq_item_export*, estos puertos están declarados dentro de la clase extendida y usan estos métodos:

- Con *get_next_item* el drive se espera a que haya un paquete para recoger del *sequencer*, y cuando lo hay lo recoge.
- Como hasta ahora la versión no bloqueante es *try_next_item*, solo que esta vez en vez de devolver un 0 o un 1 devuelve *null* si no hay nada o el paquete si lo hay.
- Para terminar la comunicación entre el *sequencer* y el *driver* para que el *sequencer* levante el bloqueo se usa *item_done*, debe usarse después de un *get_next_item* o de un *try_next_item* exitoso.
- Para que el drive devuelva un paquete de respuesta se puede hacer con el típico *put* salvo que la de esta versión no es bloqueante.

Se puede ver en la clase que se extiende de *uvm_driver* y *uvm_sequencer* como están declarados estos puertos.

```
class uvm_driver #(type REQ=uvm_sequence_item, type RSP=REQ) extends uvm_component;

  uvm_seq_item_pull_port #(REQ, RSP) seq_item_port;


class uvm_sequencer #(type REQ=uvm_sequence_item, RSP=REQ) extends
uvm_sequencer_param_base #(REQ, RSP);
                          […]

  uvm_seq_item_pull_imp #(REQ, RSP, this_type) seq_item_export;
```

2.3.20 Secuenciador.

Como ya he mencionado antes el *sequencer*, el ultimo componente del agente, es el que se encarga de enviar los paquetes generados por una secuencia al driver. Se comunica con el drive a través de los puertos TLM mencionados en el *driver*. Los paquetes generados por las *sequences* se guardan en una fifo que tiene el *sequencer* y los va enviado uno a uno al driver. Para crear esta clase se extiende de *uvm_sequencer* y esta deriva de *uvm_sequencer_param_base* y esta de otra más antes de llegar a la clase típica de *uvm_component.*

2.3.21 Secuencias.

Los *sequences* son las que crean los paquetes iniciales que se envían a través del agente hasta el DUT, estos paquetes tienen las señales configuradas de diferentes formas para ir comprobando cada posibilidad del *test* que nos atañe. Para crear este elemento se tiene que extender de *uvm_sequence* que deriva de *uvm_sequence_item* y esta de una más antes de llegar a *uvm_object.*

El flujo de información empieza con la *task body* en la secuencia, esta empieza con el método *start_item* que indica al *sequencer* que está preparando un paquete para enviarle, este bloquea el envío al drive hasta tener listo el paquete. Después la secuencia se dedica a crear el paquete, normalmente las señales son *random* y se usara el método *randomize()* para crear los valores del paquete, una vez creado el paquete con el método *finish_item()* se envía a la fifo del *sequencer*, el *sequence* se espera la respuesta y el *sequencer* desbloquea él envió de información al driver. El driver recibe el paquete del *sequencer* con *get_next_item()* o *try_next_item()* y se dedica a introducir los datos del paquete al DUT con la interfaz. Cuando el drive termina con *item_done()* le avisa al *sequencer* para desbloquear la *sequence* enviado una respuesta.

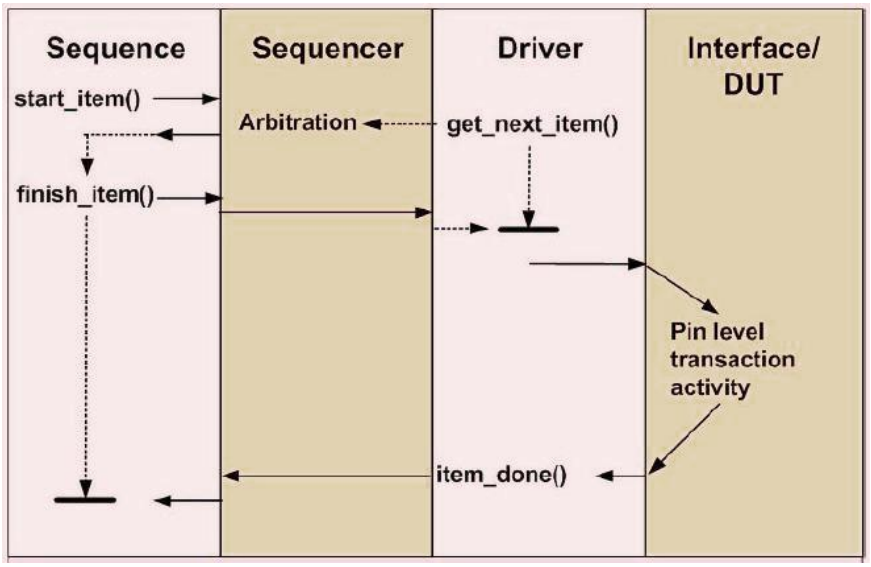


Figura 14. Envío de un elemento de secuencia

2.3.22 Items de secuencias.

El *item* del *sequence* son los paquetes que se envían a nivel de transacción, en él se definen todas las señales que se van a ir pasando por el agente y el resto de los componentes del *environment* en las diferentes transacciones, normalmente estas señales están creadas como *randoms* con *constraints* para ir generando valores diferentes en cada paquete e intentar ver todas las posibilidades, pero también se pueden crear métodos que luego usa la *sequence* para generar valores de cierta manera. Para manipular correctamente las señales colocadas en el *item* en necesario de la creación de unos métodos donde estén todas estas señales, *do_copy*, *do_compare*, *do_print* y *convert2string*, pero con la macro *uvm_field_int* se pueden crear estos métodos automáticamente con las señales que se quiera. Para crear este componente se tiene que extender la clase *uvm_sequence_item* que deriva de *uvm_transaction* y esta su vez de *uvm_object*.

2.3.23 Secuencias virtuales

La secuencia virtual es un organizador de secuencias, puede iniciar varias secuencias simultáneamente y dirigirlas a su *sequencer* correspondiente. Se usa más cuando hay más de un agente activo por lo que se hará uso de más de una secuencia para los agentes, por ejemplo, con un DUT con dos interfaces. Se encargará de coordinar y sincronizar los envíos de paquetes de las secuencias a los *sequencers* de los dos agentes para crear correctamente los diferentes casos. También se pueden usar para ejecutar más de una secuencia en un agente ordenando que secuencia actuara primero.

Se pueden crear secuencias virtuales de dos formas:

- La secuencia virtual contiene etiquetas con los *sequencers* de los agentes y va conectando las secuencias mediante estas etiquetas.
- Instanciando la secuencia virtual en un *sequencer* virtual que este conecta la secuencia virtual con el *sequencer* del agente.

2.3.24 Scoreboard

El *scoreboard* es el que verifica y comprueba que el diseño es correcto. Compara las señales que el monitor del agente ha leído del DUT a través de la interfaz, con los resultados esperados del diseño calculados dentro del *scoreboard* o en un componente exterior, llamado modelo de referencia que extiende *uvm_subscriber*, este conecta un puerto al agente para recibir los mismos datos que el DUT. El *scoreboard* recibe los datos de los monitores y del modelo de referencia a través de puertos *analysis* TLM. Para crear este componente se debe extender la clase *uvm_scoreboard* que deriva de *uvm_component*.

Los paquetes que se envían al DUT también se pueden enviar al *scoreboard* si es necesario para comparar, o si este calcula los resultados desde dentro, si los resultados esperados no son calculados desde dentro del *scoreboard*, recibirá un paquete con los datos predichos según el diseño del DUT a partir de un componente externo. Al otro lado, el monitor le enviara los resultados dados por el DUT para comparar con los datos precedidos.

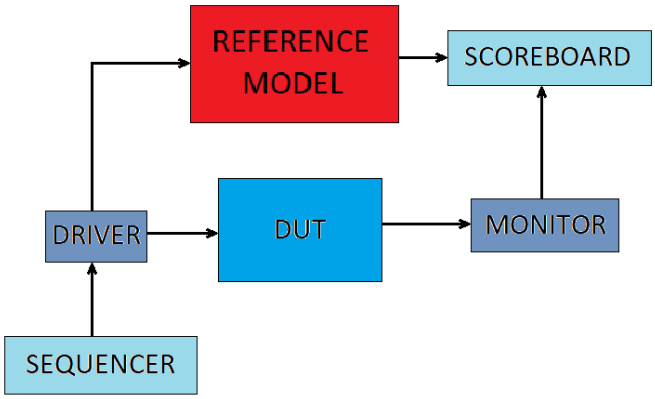


Figura 15. Modelo con ref model de verificación funcional

2.3.25 Cobertura funcional

La cobertura funcional es el porcentaje de posibilidades que se han comprobado en los casos cubiertos por el *test*. Los rangos de valores que se comprueban y su distribución son definidos por el usuario mediante *covergroups* que a su vez contienen puntos de cobertura como variables llamadas *bins*. Si se da el valor o rango de valores del *bin* este activa un evento que actualiza el porcentaje de cobertura para saber que se ha cumplido. Con esto podemos mantenernos informados de la cantidad de posibilidades que se han dado, en una señal *random* con *constraints*, por ejemplo, para asegurar su funcionamiento y darle más peso aquellas posibilidades más conflictivas que queramos observar. En otros tipos de arquitecturas de verificación se usa la cobertura funcional como señal para saber cuando pasar de caso para asegurar un porcentaje de cobertura, pero en UVM la cobertura funcional es puramente informativa.

Para crear esta clase se debe extender *uvm_subscriptor* que deriva de *uvm_component*. Este componente a diferencia del resto su extendido no es tan obvio ya que es parte de los consumidores del puerto *analysis* del monitor, de esta forma recibirá todos los datos del paquete que recibirá el *dut* para su análisis. Los datos a los que se le realiza la cobertura funcional suelen ser los de entrada al *dut* pero se puede conectar cualquier agente para su evaluación.

Capítulo 3. Desarrollo

3.1 Introducción

En este desarrollo se explicará todo el proceso a lo largo de la realización de las tareas.

3.2 Primer acercamiento

Se crea el proyecto en la herramienta "**Assistant**" indicando que se usará las librerías UVM 1.1d. También están las librerías UVM 1.2 pero a día de la realización de este proyecto tiene algunos problemas de funcionamiento, después hice la interfaz que conectara los módulos con las clases de UVM, los *modport* de está interfaz está construida con *clocking* blocks para evitar problemas de meta-estabilidad en los flancos de reloj.

```
`timescale 1ns/1ps
interface multi_if(
  input bit clk,
  input bit rst);
  logic start;
  logic fin_multi;
  logic signed[7:0]A,B;
  logic signed [15:0]salida;

  clocking mod @(posedge clk);
    input #1ns start;
    input #1ns fin_multi;
    input #1ns A;
    input #1ns B;
    input #1ns salida;
  endclocking:mod

  clocking masd @(posedge clk);
    output #2ns start;
    input #2ns fin_multi;
    output #2ns A;
    output #2ns B;
    input #2ns salida;
  endclocking:masd

  modport monitor (clocking mod);
  modport test (clocking masd);
  modport duv (
    input clk,
    input rst,
    input start,
    output fin_multi,
    input A,
    input B,
    output salida);


endinterface : multi_if
```

Para implementar el DUT primero hay que incluir todos los módulos que se usan hasta el top del DUT y después crear un último modulo que conectara la interfaz con las entradas y salidas del DUT.

```
module duv (multi_if.duv bus);

multipli multi_Booth(
  .Clock(bus.clk),
  .Reset(bus.rst),
  .Start(bus.start),
  .Fin_Mult(bus.fin_multi),
  .A(bus.A),
  .B(bus.B),
  .S(bus.salida));
```

```
endmodule
```

Ahora use las diferentes plantillas de las librerías en un orden determinado para evitar problemas con los nombres de los diferentes elementos, estas plantillas esta ubicadas en la dirección *_Templates>uvm_templates_v2>uvm_detailed_v2*.

3.2.1 La primera plantilla es TOP

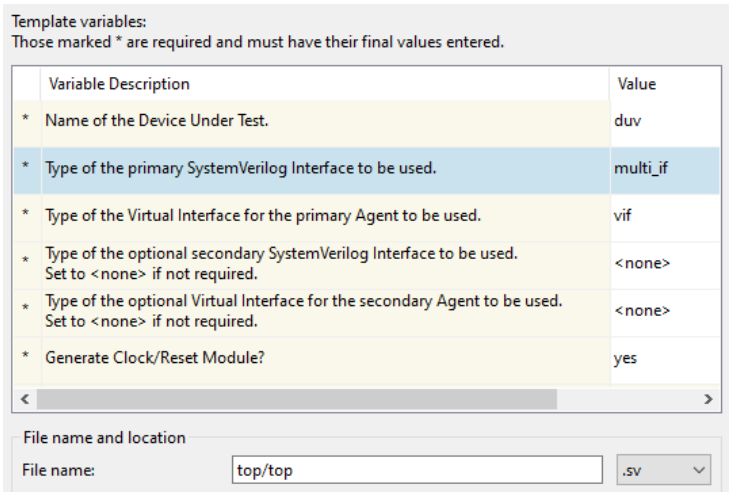


Figura 1.Parámetros de plantilla TOP

El código generado en top de la instanciación del DUT no está correcto ya que nosotros creamos un módulo para interconectar la interfaz, hay que cambiar las entradas y salidas por la interfaz de conexión del módulo.

```
duv dut (.bus(vif));
```

Hay que tener en cuenta de que en esta plantilla y en otras más adelante el *reset* está configurado como activo a nivel alto, se revisó cualquier parte de código generado en el que aparezca la variable *rst* y se invirtió sus valores si hacía falta ya que el *reset* es a nivel bajo, en este caso en el código de *clock_reset* hay *reset* que se cambió.

3.2.2 La segunda plantilla es AGENT_complete.

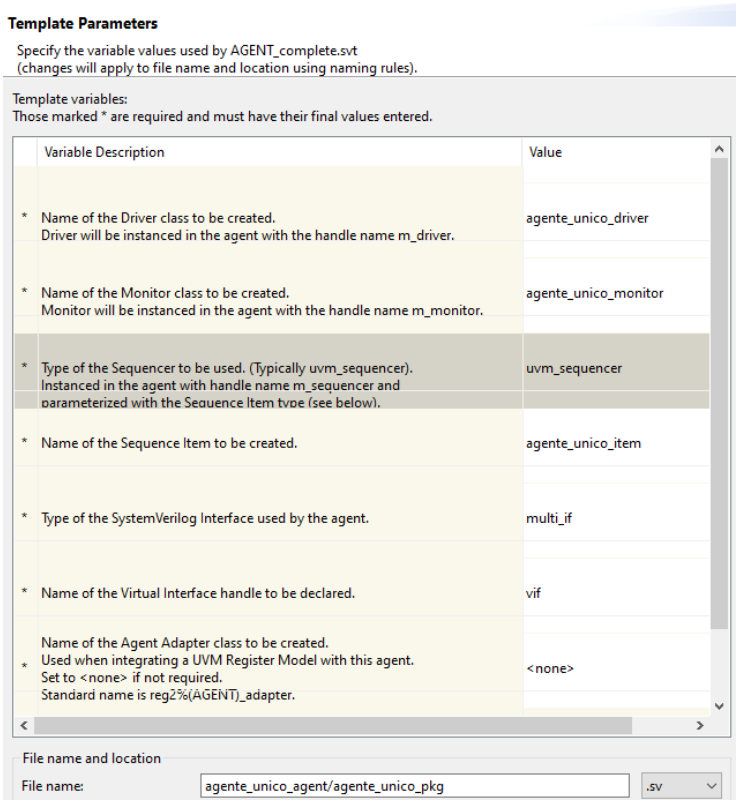


Figura 2.Parámetros de plantilla AGENT_complete

Esta plantilla tiene la opción de crear dos agentes para una variación de la arquitectura UVM, pero no se usó esta opción y se trabajó con un agente único.

Esta plantilla genera mucho código y el primero que se cambio fue el agente *item*, en este se añadió todas las variables de entrada y salida del módulo además de convertir las variables de entrada en *randoms* circulares con un par de *constrains*.

```
class agente_unico_item extends uvm_sequence_item;

  // user stimulus variables (rand)
  bit start;
  randc bit signed [7:0] A;
  randc bit signed[7:0] B;
  constraint p_p {A[7]==0; B[7]==0;};
  constraint p_n {A[7]==0; B[7]==1;};
  constraint n_p {A[7]==1; B[7]==0;};
  constraint n_n {A[7]==1; B[7]==1;};
  // user response variables (non rand)
  bit fin_multi;

  bit signed [15:0] S;
```

Además, se añadió las variables en las distintas funciones para poder usarlas:

- *do_copy* para poder traspasar entre distintas estancias.

```
virtual function void do_copy(uvm_object rhs);
    agente_unico_item rhs_;

    if(!$cast(rhs_, rhs)) begin
      `uvm_error({get_type_name(),":do_copy"}, "Copy Failed, type mismatch")
      return;
    end

    super.do_copy(rhs);
    start=rhs_.start;
```

```
    A=rhs_.A;
    B=rhs_.B;
    fin_multi=rhs_.fin_multi;
    S=rhs_.S;


  endfunction: do_copy
```

- *do_compare* compara las variables de 2 estancias, esto es para la comparación en *scoreboard*.

```
virtual function bit do_compare(uvm_object rhs, uvm_comparer comparer);
    agente_unico_item rhs_;

    if(!$cast(rhs_, rhs)) begin
      `uvm_error({get_type_name(),":do_compare"}, "Compare Failed, type mismatch")
      return 0;
    end

    return((super.do_compare(rhs, comparer))

      && (start==rhs_.start)
      && (A==rhs_.A)
      && (B==rhs_.B)
      && (fin_multi==rhs_.fin_multi)
      && (S==rhs_.S)
      );


  endfunction: do_compare
```

- *convert2string* transforma en *string* las variables para mostrar en la caja de comando del simulador.

```
function string convert2string();
    string str;
    str = super.convert2string();
    $sformat(str, "%s\n",str);
    $sformat(str, "%s start: \t%0d\n",str, start);
    $sformat(str, "%s A: \t%0d\n",str, A);
    $sformat(str, "%s B: \t%0d\n",str, B);
    $sformat(str, "%s fin_multi: \t%0d\n",str, fin_multi);
    $sformat(str, "%s S: \t%0d\n",str, S);
    return str;

  endfunction: convert2string
```

En el ag*ent_driver* hay que seleccionar en la *task run_phase* si la puesta de datos tiene que parar la simulación, para ello hay que indicar si es *blocking* o *non_blocking*, en este caso se seleccionó *blocking* para saber exactamente cuándo ocurrirá en la simulación este proceso y evitar problemas.

También hay que añadir en la *task drive_dut* como se introducirán estas señales en el DUT.

```
virtual task drive_dut();
  // Drive transactions onto interface signals
    `uvm_info({get_type_name(),":drive_dut"},req_txn.convert2string(),UVM_MEDIUM)

    vif.masd.start <= 1;
    vif.masd.A <= req_txn.A;
    vif.masd.B <= req_txn.B;
    repeat(3) @(vif.masd);
    vif.masd.start <= 0;
    @(negedge vif.masd.fin_multi);

  endtask: drive_dut
```

En esta clase hay un caso *reset* en las *task* de *blocking* y *non_blocking* siendo que solo se usa una solo se cambió la de *blocking*.

Otro código que se cambio fue el de agente monitor, en este se añade como se capturaran las variables y en que momento se hará.

```
task monitor_dut();
  // Monitor transactions from the interface
    forever begin
      @(vif.mod);
      //if (vif.rst) begin
        if (vif.mod.fin_multi == 1) begin
          // mon_txn.data = vif.data;
          mon_txn.S = vif.mod.salida;
          mon_txn.A = vif.mod.A;
          mon_txn.B = vif.mod.B;
          $cast(t, mon_txn.clone());
          ap.write(t);
          `uvm_info({get_type_name(),":monitor_dut"}, t.convert2string(), UVM_MEDIUM)
        end
      end

  endtask: monitor_dut
```

Como se puede ver la captura ocurre en la activación del *modport* mod que en este caso según la interfaz es en flanco de subida de reloj y sólo si la variable *fin_multi* está en 1, cuando se cumplen las condiciones captura todas las variables.

3.2.3 La tercera plantilla es ENV_complete

En esta plantilla es importante que el quinto parámetro sea THREADED que es el de las FIFOs.

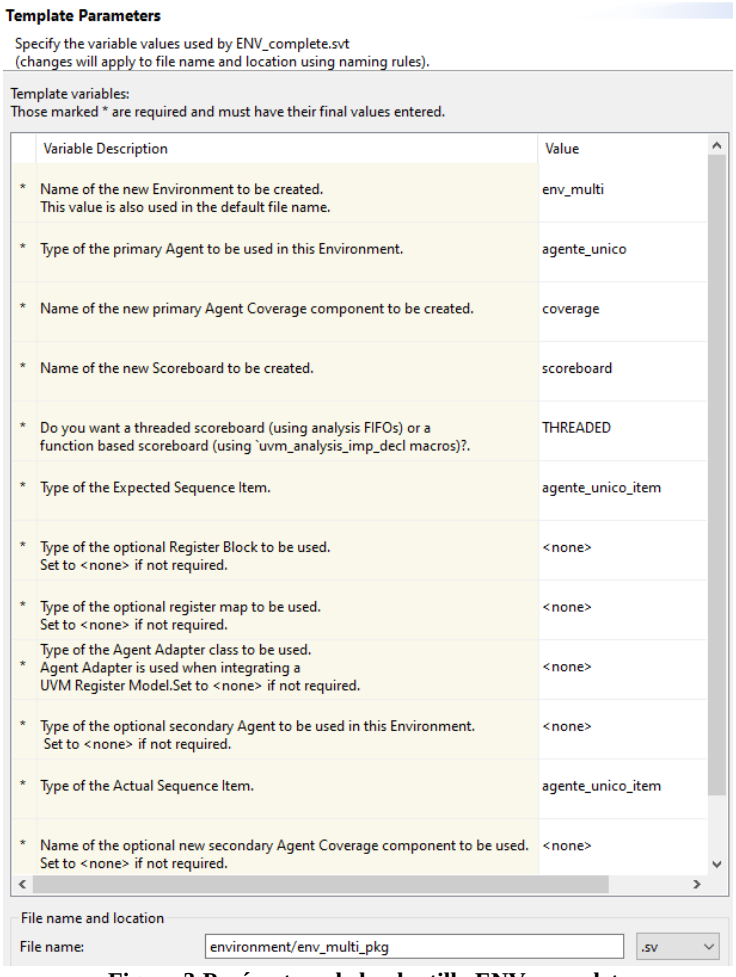


Figura 3.Parámetros de la plantilla ENV_complete

En el código generado se cambió una función del código *env_multi*, en *connect_fase* se cambió el *expected_export* por el *actual_export* que le corresponde al *scoreboard*.

```
if(m_cfg.has_scoreboard) begin
      m_agente_unico.ap.connect(m_scoreboard.actual_export);
    end
```

El código *coverage* también se cambió, como su nombre indica es donde se añade la cobertura funcional.

```
    covergroup cov_item_cover;
      CPA1: coverpoint cov_item.A;
      CPB1: coverpoint cov_item.B;
      CP: cross CPA1,CPB1;

    endgroup
```

Por último, se cambió el *scoreboard*, la comparación en el *scoreboard* es el *predicted* con el *actual*, pero el *predicted* no se usa en este método por lo que hay que cambiarlo y eliminarlo de las notificaciones.

```
virtual task run_phase(uvm_phase phase);

    // --- Example code ---
    forever begin
      expected_fifo.get(expected_txn);
      actual_fifo.get(actual_txn);
      predicted_txn = predict_expected(expected_txn);
      assert (expected_txn.compare(actual_txn)) begin
        $sformat(msg, "Expected RSP: {%s}  Actual RSP: {%s}",
                      expected_txn.convert2string(),
                      actual_txn.convert2string());
        `uvm_info("SCOREBOARD_PASS", msg, UVM_MEDIUM)
      end
      else begin
        $sformat(msg, "Expected RSP: {%s}  Actual RSP: {%s}",
                      expected_txn.convert2string(),
                      //predicted_txn.convert2string(),
                      actual_txn.convert2string());
        `uvm_error("SCOREBOARD_FAIL_", msg);
      end
    end
    // --------------------


  endtask: run_phase
```

3.2.4 *La cuarta plantilla es SEQUENCE_complete*

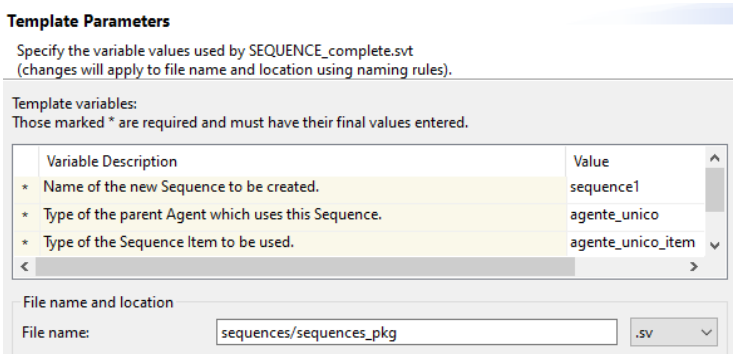


Figura 4.Parámetros de la plantilla SEQUENCE_complete

En el código de sequence1 se puede cambiar el número de repeticiones que harán en la secuencia.

En sequence1 también es donde se elige que *constrains* estarán activos a la hora de hacer la función *randomize*.

```
task body();
    req_item_h = agente_unico_item::type_id::create("req_item_h");
    repeat(num_repetions) begin
      start_item(req_item_h);
      req_item_h.p_p.constraint_mode(1);
      req_item_h.p_n.constraint_mode(0);
      req_item_h.n_p.constraint_mode(0);
      req_item_h.n_n.constraint_mode(0);
      req_item_h.ct="p_p";
      assert(req_item_h.randomize());
      `uvm_info({get_type_name(),":body"},{"Sending transaction
",req_item_h.convert2string()}, UVM_MEDIUM)

      finish_item(req_item_h);
       end
```

Esta parte del código tiene más como este código activando y desactivando distintos *constrains*.

3.2.5 La quinta plantilla es VSEQUENCE_complete

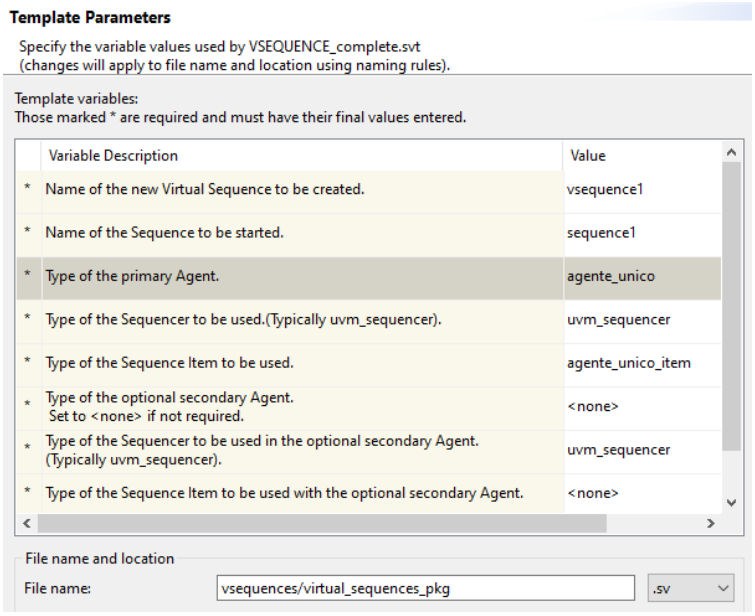


Figura 5.Parámetros de la plantilla VSEQUENCE_complete

3.2.6 *La sexta plantilla es TEST_complete*

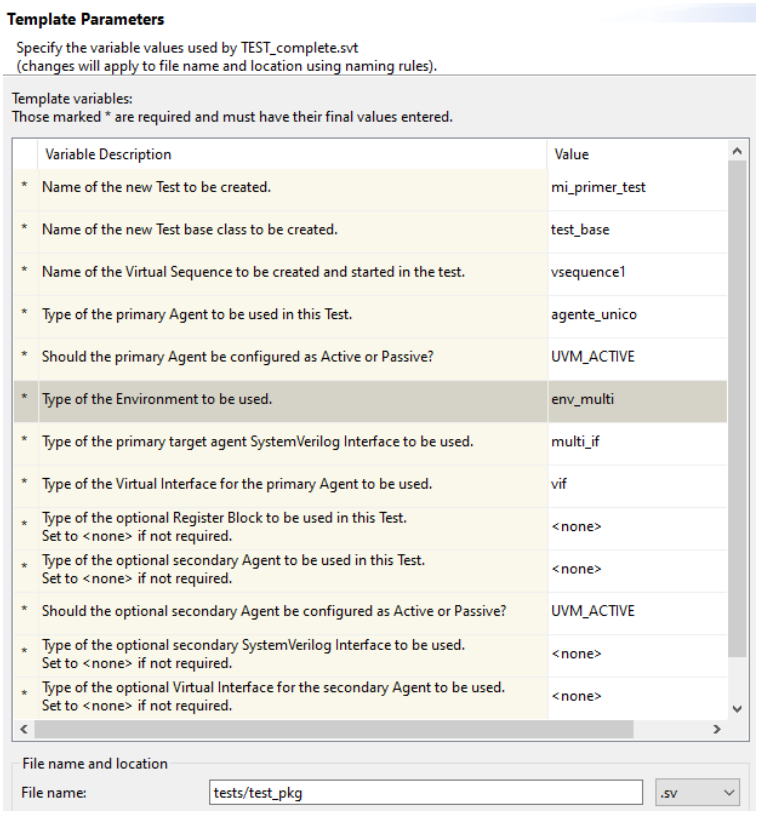


Figura 6.Parámetros de la plantilla TEST_complete

3.2.7 *La séptima plantilla es SCOREBOARD_PREDICTOR*

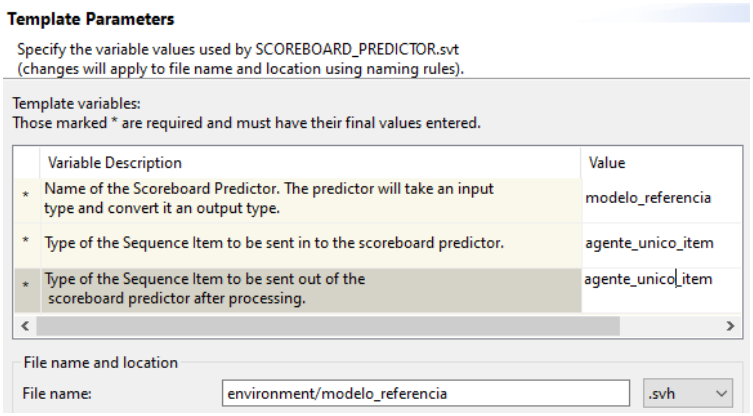


Figura 7.Parámetros de la plantilla SCOREBOARD_PREDICTOR

Esta plantilla no entra dentro de las configuraciones iniciales por lo que se modificó parte del código para hacerla funcionar.

Uno de los códigos que se cambio es el *env_multi_pkg*, añadiéndolo en los *include*.

```
// Include the components
  `include "coverage.svh"
  `include "scoreboard.svh"
  `include "modelo_referencia.svh"
  `include "env_multi.svh"
```

Es importante que esté antes del *include env_multi* si no tendrá problemas para reconocerlo en *env_multi*.

El siguiente código que se cambio es el *env_multi*, se instancio el predictor en el código arrastrando la clase creada del navegador de proyecto.

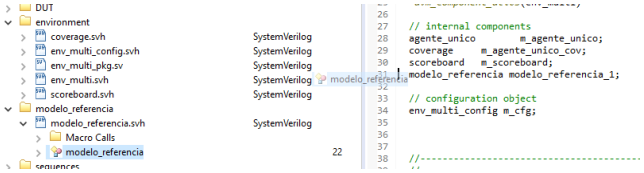


Figura 8.Instanciación de la clase modelo_referencia

Si se hace de esta forma el código actualizara sus funciones con la nueva instanciación automáticamente.

En la función *connect_phase* se añadió en el paréntesis el *expected_export* del *scoreboard*, además se colocó una línea más que conecta lo añadido con las diferencias instancias de agente.

```
m_agente_unico.ap.connect(modelo_referencia_1.analysis_export);
modelo_referencia_1.m_output_ap.connect(m_scoreboard.expected_export);
```

Ahora se añadió en el código generado por la plantilla el modelo de referencia que se usara para compararlo con el DUT, en este caso como es solo un multiplicador el modelo es muy sencillo.

```
function void write(T t);
    agente_unico_item   m_out_item;
    m_out_item = agente_unico_item::type_id::create("m_out_item");

    //t is the input sequence item (transaction).  Process t and then
    // write the new processed output to the m_output_ap.
    m_out_item.S=t.A*t.B;
    m_out_item.A=t.A;
    m_out_item.B=t.B;

    m_output_ap.write(m_out_item);

  endfunction : write
```

Con esta última plantilla ya se podría simular desde el navegador de jerarquía dándole al test con el nombre que se puso después de haber compilado sin ningún error

3.3 Banco de pruebas para RISC-V

Los pasos a seguir al principio son los mismo que en el multiplicador por lo que explicare las diferencias y lo que se hizo después. El módulo a comprobar debe tener dos outputs extras que normalmente no tiene, el control de escritura en el registro y el dato a escribir en el registro.

3.3.1 Diferencia de variables

Este banco de pruebas al ser otro DUT sus variables son diferentes, como luego nos extenderemos más en el desarrollo primero se dejará claro que variables, *monitor*, *constraints*, *covergroups*, interfaz y los nombres de los códigos tiene este diseño.

- Nombres: env_multi = env_RISC_V, modelo_referencia = my_TinuC, sequence1 = sequences, mi_primer_test = start_test, agente_unico = unico, vsequence1 = vsequences, multi_if = RISC_V_interface.

- Interfaz

```
`timescale 1ns/1ps
interface RISC_V_interface(
  input bit clk,
  input bit rst);
  bit [31:0] idata;
  bit [31:0] ddata_r;
  bit [31:0] iaddr;
  bit [31:0] daddr;
  bit [31:0] ddata_w;
  bit [31:0] reg_data;
```

```
  bit d_rw;
  bit enable_reg_write;
  string instru;

  clocking mod @(posedge clk);
    default input #1ns;
    input idata ,ddata_r, iaddr, daddr, ddata_w, reg_data,d_rw, enable_reg_write;
  endclocking:mod

  clocking masd @(posedge clk);
    default input #2ns output #2ns;
    output idata ,ddata_r;
    input iaddr, daddr, ddata_w, reg_data, d_rw, enable_reg_write;

  endclocking:masd

  modport monitor (clocking mod);
  modport test (clocking masd);
  modport duv (
    input clk,
    input rst,
    input idata ,
    input ddata_r,
    output iaddr,
    output daddr,
    output ddata_w,
    output reg_data,
    output d_rw,
    output enable_reg_write);


endinterface : RISC_V_interface
```

- Variables y constrains

```
// user stimulus variables (rand)
  randc bit [31:0] idata;
  rand bit [31:0] ddata_r;
  constraint data {ddata_r != 0;}
  constraint addi {idata[6:0] == 7'h13; idata[14:12] == 3'h0;}
  constraint slti {idata[6:0] == 7'h13; idata[14:12] == 3'h2;}
  constraint sltiu {idata[6:0] == 7'h13; idata[14:12] == 3'h3;}
  constraint andi {idata[6:0] == 7'h13; idata[14:12] == 3'h7;}
  constraint ori {idata[6:0] == 7'h13; idata[14:12] == 3'h6;}
  constraint xori {idata[6:0] == 7'h13; idata[14:12] == 3'h4;}
  constraint lui {idata[6:0] == 7'h37;}
  constraint auipc {idata[6:0] == 7'h17;}
  constraint add {idata[6:0] == 7'h33; idata[14:12] == 3'h0; idata[31:25] == 7'h00;}
  constraint slt {idata[6:0] == 7'h33; idata[14:12] == 3'h2; idata[31:25] == 7'h00;}
  constraint sltu {idata[6:0] == 7'h33; idata[14:12] == 3'h3; idata[31:25] == 7'h00;}
  constraint and_ {idata[6:0] == 7'h33; idata[14:12] == 3'h7; idata[31:25] == 7'h00;}
  constraint or_ {idata[6:0] == 7'h33; idata[14:12] == 3'h6; idata[31:25] == 7'h00;}
  constraint xor_ {idata[6:0] == 7'h33; idata[14:12] == 3'h4; idata[31:25] == 7'h00;}
  constraint sub {idata[6:0] == 7'h33; idata[14:12] == 3'h0; idata[31:25] == 7'h20;}
  constraint beq {idata[6:0] == 7'h63; idata[14:12] == 3'h0;}
  constraint bne {idata[6:0] == 7'h63; idata[14:12] == 3'h1;}
  constraint lw {idata[6:0] == 7'h03; idata[14:12] == 3'h2; idata[11:7] ==
const'(idata[11:7]);}
  constraint sw {idata[6:0] == 7'h23; idata[14:12] == 3'h2;}
  constraint slli {idata[6:0] == 7'h13; idata[14:12] == 3'h1; idata[31:25] == 7'h00;}
  constraint srli {idata[6:0] == 7'h13; idata[14:12] == 3'h5; idata[31:25] == 7'h00;}
  constraint srai {idata[6:0] == 7'h13; idata[14:12] == 3'h5; idata[31:25] == 7'h20;}
  constraint sll {idata[6:0] == 7'h33; idata[14:12] == 3'h1; idata[31:25] == 7'h00;}
  constraint srl {idata[6:0] == 7'h33; idata[14:12] == 3'h5; idata[31:25] == 7'h00;}
  constraint sra {idata[6:0] == 7'h33; idata[14:12] == 3'h5; idata[31:25] == 7'h20;}
  constraint jal {idata[6:0] == 7'h6f;}
  constraint jalr {idata[6:0] == 7'h67; idata[14:12] == 3'h0;}
```

```
  constraint blt {idata[6:0] == 7'h63; idata[14:12] == 3'h4;}
  constraint bltu {idata[6:0] == 7'h63; idata[14:12] == 3'h6;}
  constraint bge {idata[6:0] == 7'h63; idata[14:12] == 3'h5;}
  constraint bgeu {idata[6:0] == 7'h63; idata[14:12] == 3'h7;}
  // user response variables (non rand)
  bit [31:0] iaddr;
  bit [31:0] daddr;
  bit [31:0] ddata_w;
  bit [31:0] reg_data;
  bit d_rw;
  bit enable_reg_write;
  string instru;
  static bit error = 0;
  static bit [30:0] errorlist = 0;
  enum bit[4:0]{_addi, _slti, _sltiu, _andi, _ori, _xori, _lui, _auipc, _add, _slt,
_sltu, _and, _or, _xor, _sub, _beq, _bne, _lw, _sw, _slli, _srli, _srai, _sll, _srl,
_sra, _jal, _jalr, _blt, _bltu, _bge, _bgeu} check;
```

- *Covergroup*, al ser muchos bits se hacía demasiado grande hacer unos *covergroups* exhaustivos por lo que al final se optó por simplificarlos, aunque no fuera tan preciso.

```
  covergroup cov_item_cover;
    opcode: coverpoint cov_item.idata[6:0]{
      wildcard bins opcode[] = {7'b0xx0011,
                                7'b0x10111,
                                7'b110x111,
                                7'b1100011};
    }
    rd: coverpoint cov_item.idata[11:7];
    fun3: coverpoint cov_item.idata[14:12];
    rs1: coverpoint cov_item.idata[19:15];
    rs2: coverpoint cov_item.idata[24:20];
    fun7: coverpoint cov_item.idata[31:25]{bins all[] = {[0:$]};}
    imm:cross rd, fun3, rs1, rs2, fun7;

  endgroup
```

- Monitor

```
  task monitor_dut();
  // Monitor transactions from the interface
    forever begin
      @(vif.mod);
     // if (vif.rst) begin
        if (mon_txn.iaddr!=vif.mod.iaddr) begin
          mon_txn.idata = vif.mod.idata;

          mon_txn.ddata_r = vif.mod.ddata_r;
```

No esta toda la función del monitor ya que son todas las variables otra vez y lo que quiero destacar es el hecho de que el monitor se ejecuta cada vez que el valor de PC cambia.

3.3.2 Creación e integración del gold_model

En este apartado se explicará todo lo que se hizo hasta que se pudo comparar el *dut* actual con el *gold_model* que se había elegido.

Teniendo en cuanto que el banco lo iban a ver estudiantes que podrían ver el *gold_model* se usó el archivo de modulo *gate_level* que crea "**Quartus**" en vez del módulo RTL, pero para que este archivo funcionara se necesitaban las librerías de altera y el archivo *.sdo* que crea "**Quartus**" en la carpeta de "**QuestaSim**" que está en la carpeta *_Build_Files* del proyecto.

Para facilitar añadir las librerías se actualizo la herramienta "**Assistant**" a la versión 2022.2 ya que esta tenía soporte para conectarse a "**Quartus**" y por lo tanto coger las librerías a través de este.

Para hacer la conexión se usó la opción compilar librerías de simulación en la pestaña de "**Quartus**" y se eligió un directorio de compilación dentro del proyecto con una carpeta nueva que se llamó *libraries*.

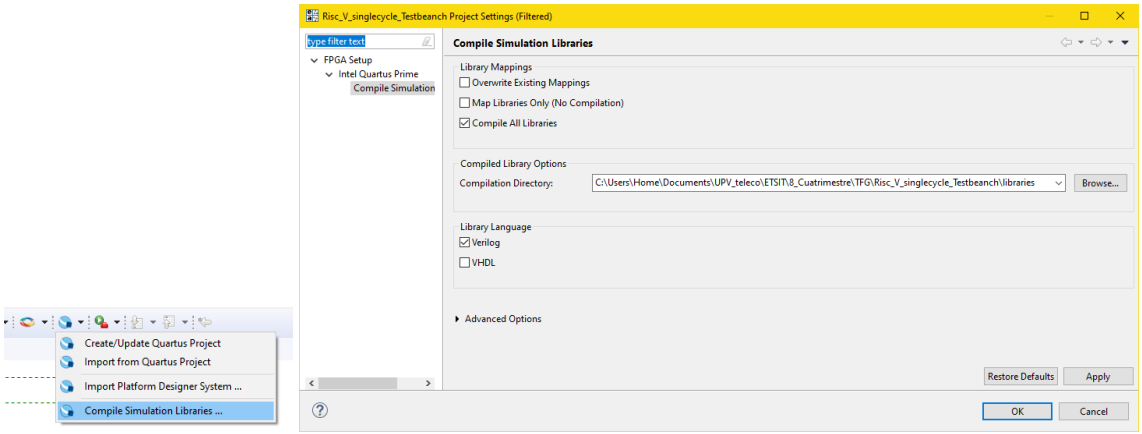


Figura 9.Ventana de Compile Simulation Libraries.

Ahora que las librerias estan compiladas hace falta decirle a "QuestaSim" que las carge antes de hacer nada y para eso se cambió el comando vsim inicial que esta en project settings.

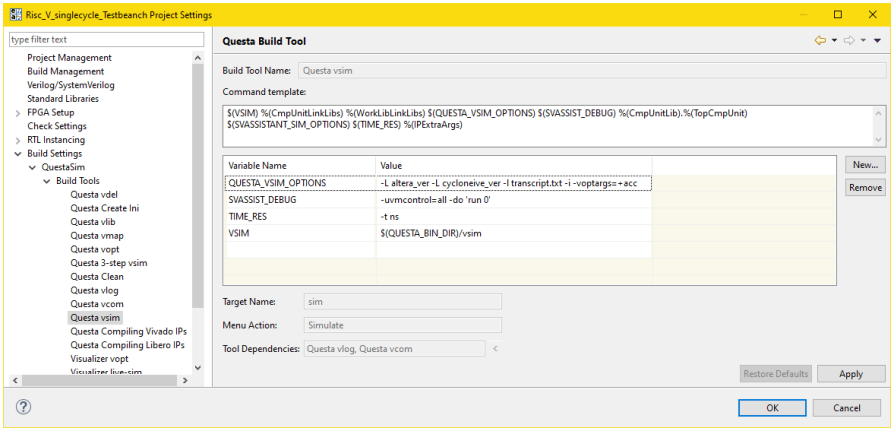


Figura 10. Ventana de Proyect Settings

Teniendo el modulo del gold_model preparado se creo un top del gold_model llamado modelo_referencia en donde se instancia con interfaz el gold_model.

```
module modelo_referencia (RISC_V_interface.duv bus);

RVI32 RISC_V (
      .CLK(bus.clk),
      .RESET_N(bus.rst),
      .instruction(bus.idata),
      .ROMADDRESS(bus.iaddr),
      .RAMADDRESS(bus.daddr),
      .W_DATARAM(bus.ddata_w),
      .DATARAM(bus.ddata_r),
      .MemWrite(bus.d_rw),
      .RegWrite(bus.enable_reg_write),
      .WRITE_DATAREG(bus.reg_data));

endmodule
```

Teniendo esto se instancio en el top junto con una nueva instancia de interfaz.

```
  RISC_V_interface     vif(clk, rst);
  RISC_V_interface    mr(clk, rst);
```

```
  clock_reset cr (clk, rst);

  dut  dut (.bus(vif.duv));


  modelo_referencia modelo_referencia(.bus(mr.duv));
```

Para que esta nueva instancia de interfaz funcione hay que virtualizarla y para ello se añadió un nuevo *uvm_config* junto con el de la anterior instancia.

```
    uvm_config_db #(virtual RISC_V_interface)::set(null, "uvm_test_top", "vif_vif",
vif);

    uvm_config_db #(virtual RISC_V_interface)::set(null, "uvm_test_top", "mr_vif", mr);
```

También se añadió un *if* en *test_base* igual al de la primera instancia para conectar la virtualización a *unico_config*.

```
if(!uvm_config_db #(virtual RISC_V_interface)::get(this, "", "vif_vif",
m_unico_cfg.vif))
    begin
      `uvm_error("RESOURCE_ERROR", "RISC_V_interface virtual interface not found")
    end
    if(!uvm_config_db #(virtual RISC_V_interface)::get(this, "", "mr_vif",
m_unico_cfg.mr))
    begin
      `uvm_error("RESOURCE_ERROR", "RISC_V_interface virtual interface not found")

    end
```

Para poder meter valores desde *unico_drive* hay que conectar la interfaz virtual nueva con alguna ya creada y como el único que tiene la virtualización es *unico_config*, desde *unico* se conectó la interfaz virtual de *config* con la de drive en *connect_phase* al igual que la primera instancia de interfaz.

```
virtual function void connect_phase(uvm_phase phase);

    // Monitor is always connected
    m_monitor.ap.connect(ap);
    m_monitor.vif = m_cfg.vif;
    // Driver and Sequencer only connected if agent is active
    if (m_cfg.is_active == UVM_ACTIVE) begin
      m_driver.vif = m_cfg.vif;
      m_driver.mr = m_cfg.mr;
      m_driver.seq_item_port.connect(m_sequencer.seq_item_export);
    end


  endfunction: connect_phase
```

Ahora ya se puede meter valores en el DUT y *modelo_referencia* al mismo tiempo.

Mientras que el DUT se comprueba en *unico_monitor*, el *modelo_referencia* se tiene que comprobar en *my_TinuC*, por lo que hay que conectar una nueva interfaz virtual a una existente, la conexión se debe realizar en la función *connect_phase* de *env_RISC_V*, hay varias maneras de conectarla a una ya existente, en este proyecto se optó por conectarla a la interfaz virtual de *unico_config* a través de la instancia de *env_RISC_V_config*.

```
  my_TinuC_1.mr = m_cfg.m_unico_cfg.mr;
  endfunction: connect_phase
```

Otro camino que podría ser es crear una nueva interfaz virtual en *env_RISC_V* y conectarla a la de *unico_config* a través de la función *connect_phase* de *test_base*, después esta nueva interfaz virtual se conectaría a la de *my_TinuC* en el mismo sitio que el primer camino.

Con esto el *gold_model* ya se puede comparar con el DUT además cuando se ejecuta el monitor se ejecuta el modelo de referencia al mismo tiempo por lo que la lectura de datos es simultánea.

3.3.3 Comparación

Al momento de realizar la comparación hay que tener varias cosas en cuenta:

- Si no se está escribiendo o leyendo de memoria da igual que dirección o dato haya en memoria.
- Si no está escribiendo en memoria también da igual que dato se podría escribir en memoria.
- Si no está escribiendo en registros da igual que dato se podría escribir en registros.
- Si no hay ningún dato en la memoria de instrucción mejor no comparar nada.

Todo esto se tuvo en cuenta ya que como no se sabe qué hará cada alumno en esas situaciones que da igual es mejor no comparar.

Teniendo todo esto en cuenta la función de comparación en *unico_item* queda así.

```
return(((super.do_compare(rhs, comparer))

      && (idata == rhs_.idata)
      && (ddata_r == rhs_.ddata_r)
      && (iaddr == rhs_.iaddr)
      && (daddr == rhs_.daddr || idata[6:0] != 7'h03 || idata[6:0] != 7'h23)
      && (ddata_w == rhs_.ddata_w || d_rw == 1'b0)
      && (d_rw == rhs_.d_rw)
      && (enable_reg_write == rhs_.enable_reg_write)
      && (reg_data == rhs_.reg_data || enable_reg_write == 1'b0))
      || (idata == 0)


      );
```

3.3.4 Secuencia

La secuencia está dividida en 3 grupos de instrucción que dependen de la cantidad de bits libres y por tanto de variabilidad de estas.

1. El primer grupo con 15 bits libres están las instrucciones R e IR.
   Las instrucciones IR no están en el estándar de RISC_V, pero se tuvo que nombrar este grupo de instrucciones por su peculiar funcionamiento, las instrucciones *slli*, *srli* y *srai* son instrucciones inmediatas que tienen fun7 por lo que limita la cantidad de bits libres que tienen.

2. El segundo grupo con 22 bits libres están las instrucciones I, S y SB.

3. El tercer grupo con 25 bits libres están las instrucciones U y UJ.

```
int num_repetionsI_S_SB = 33;
bit[4:0] i;
int num_repetionsR_IR = 33;

int num_repetionsU_UJ = 33;
```

Para un análisis correcto la secuencia tiene una primera carga de todos los registros para que las instrucciones trabajen con números y no con 0, de esta forma si el DUT tiene algún fallo aparecerá antes.

Esta carga se realizó con la instrucción *lw* la cual se le quito de su *constrain* la capacidad de aleatorizar el registro de destino para poder meterla en un bucle, al principio de la secuencia, en cada bucle escribe en un registro para rellenarlo y al llegar al bucle 33 el valor de registro de destino vuelve a valer 0.

```
repeat(33) begin
        if(!req_item_h.errorlist[req_item_h.check] && !req_item_h.error)begin
          start_item(req_item_h);
```

```
          if(!req_item_h.error)begin
            req_item_h.constraint_type("lw");
            assert(req_item_h.randomize());
            req_item_h.idata[11:7]=i;
            i++;
            `uvm_info({get_type_name(),":body"},{"Sending transaction
",req_item_h.convert2string()}, UVM_MEDIUM)
          end
          finish_item(req_item_h);
        end

      end
```

Además, hay instrucciones que pueden vaciar los registros, por ejemplo, la instrucción *slti* pone en el registro de destino un 0 o un 1 por lo que puede acabar vaciando los registros, para evitar eso después de cada instrucción que tenga este problema se pone la instrucción *lw* y como esta no varía el registro de destino el registro de destino será igual que el de la instrucción anterior.

```
       repeat(num_repetionsI_S_SB) begin
        req_item_h.check = req_item_h._slti;
        if(!req_item_h.errorlist[req_item_h.check] && !req_item_h.error)begin
          start_item(req_item_h);
          if(!req_item_h.error)begin
            req_item_h.constraint_type("slti");
            assert(req_item_h.randomize());
            `uvm_info({get_type_name(),":body"},{"Sending transaction
",req_item_h.convert2string()}, UVM_MEDIUM)
          end
          finish_item(req_item_h);
        end

        req_item_h.check = req_item_h._lw;
        if(!req_item_h.errorlist[req_item_h.check] && !req_item_h.error)begin
          start_item(req_item_h);
          if(!req_item_h.error)begin
            req_item_h.constraint_type("lw");
            assert(req_item_h.randomize());
            `uvm_info({get_type_name(),":body"},{"Sending transaction
",req_item_h.convert2string()}, UVM_MEDIUM)
          end
          finish_item(req_item_h);
        end

      end
```

Si juntamos las 33 veces del principio mes todas las demás del resto de instrucciones la instrucción *lw* es la que más veces se comprueba.

Con esto ya se ha visto la base de la secuencia ahora vienen los extras.

3.4 Propuestas

En el proyecto después de tener los básico se propusieron dos cosas, que la simulación te dijera que instrucción ha fallado y aunque falle una instrucción vuelva a empezar saltándose esa instrucción para evitar tener que quitar las instrucciones a mano, porque las instrucciones erróneas afectan al conjunto de registro que provoca malas comparaciones en las siguientes instrucciones.

3.4.1 Etiquetar instrucciones

Para realizar esta propuesta se creó una variable *string* que contiene el nombre de la instrucción que se está usando en ese momento, para evitar problemas con la FIFO la variable deberá ser capturado por el monitor, para aprovechar esta variable se usará también para controlar los *constrain* que comprimirá los bloques de la secuencia.

Lo primero que se hizo fue poner la variable *instru* en la interfaz y en *unico_item* variable que de hecho se puede ver en el código anterior. En *unico_item* se puso también en la función *do_copy* para que la variable fuera pasando por las distintas estancias de *item*.

```
super.do_copy(rhs);
    idata = rhs_.idata;
    ddata_r = rhs_.ddata_r;
    iaddr = rhs_.iaddr;
    daddr = rhs_.daddr;
    ddata_w = rhs_.ddata_w;
    reg_data = rhs_.reg_data;
    d_rw = rhs_.d_rw;
    enable_reg_write = rhs_.enable_reg_write;
    instru = rhs_.instru;
```

Para comprimir la secuencia y aprovechar la variable se creó una función que cambia los estados de los *constraints*. Esta función se nombró como *constraint_type*.

```
function void constraint_type(string ct_mode);
    data.constraint_mode(1);
    addi.constraint_mode(0);
    slti.constraint_mode(0);
    sltiu.constraint_mode(0);
    andi.constraint_mode(0);
    ori.constraint_mode(0);
    xori.constraint_mode(0);
    lui.constraint_mode(0);
    auipc.constraint_mode(0);
    add.constraint_mode(0);
    slt.constraint_mode(0);
    sltu.constraint_mode(0);
    and_.constraint_mode(0);
    or_.constraint_mode(0);
    xor_.constraint_mode(0);
    sub.constraint_mode(0);
    beq.constraint_mode(0);
    bne.constraint_mode(0);
    lw.constraint_mode(0);
    sw.constraint_mode(0);
    slli.constraint_mode(0);
    srli.constraint_mode(0);
    srai.constraint_mode(0);
    sll.constraint_mode(0);
    srl.constraint_mode(0);
    sra.constraint_mode(0);
    jal.constraint_mode(0);
    jalr.constraint_mode(0);
    blt.constraint_mode(0);
    bltu.constraint_mode(0);
    bge.constraint_mode(0);
    bgeu.constraint_mode(0);
    case(ct_mode)
      "addi": addi.constraint_mode(1);
      "slti": slti.constraint_mode(1);
      "sltiu": sltiu.constraint_mode(1);
      "andi": andi.constraint_mode(1);
      "ori": ori.constraint_mode(1);
      "xori": xori.constraint_mode(1);
      "lui": lui.constraint_mode(1);
      "auipc": auipc.constraint_mode(1);
      "add": add.constraint_mode(1);
      "slt": slt.constraint_mode(1);
      "sltu": sltu.constraint_mode(1);
      "and": and_.constraint_mode(1);
      "or": or_.constraint_mode(1);
      "xor": xor_.constraint_mode(1);
```

```
      "sub": sub.constraint_mode(1);
      "beq": beq.constraint_mode(1);
      "bne": bne.constraint_mode(1);
      "lw": lw.constraint_mode(1);
      "sw": sw.constraint_mode(1);
      "slli": slli.constraint_mode(1);
      "srli": srli.constraint_mode(1);
      "srai": srai.constraint_mode(1);
      "sll": sll.constraint_mode(1);
      "srl": srl.constraint_mode(1);
      "sra": sra.constraint_mode(1);
      "jal": jal.constraint_mode(1);
      "jalr": jalr.constraint_mode(1);
      "blt": blt.constraint_mode(1);
      "bltu": bltu.constraint_mode(1);
      "bge": bge.constraint_mode(1);
      "bgeu": bgeu.constraint_mode(1);
      default: data.constraint_mode(0);
    endcase
    instru=ct_mode;
  endfunction: constraint_type
```

Ahora para que la variable entre en la interfaz hay que añadirla con *unico_drive*, pero como la instrucción es un *string* la interfaz no permite introducirla en un *modport*, para evitar problemas de captura el traspaso de la variable a la interfaz se producirá 2ns después.

```
    vif.masd.idata <= req_txn.idata;
    vif.masd.ddata_r <= req_txn.ddata_r;
    mr.masd.idata <= req_txn.idata;
    mr.masd.ddata_r <= req_txn.ddata_r;
    #2ns vif.instru <= req_txn.instru;
```

Después solo queda un paso para tener la variable donde queremos y en el momento que queremos, en *unico_monitor* hay que añadir la variable, aunque en este caso no hace falta poner un tiempo de espera.

```
          mon_txn.reg_data = vif.mod.reg_data;
          mon_txn.d_rw = vif.mod.d_rw;
          mon_txn.enable_reg_write = vif.mod.enable_reg_write;
          mon_txn.instru = vif.instru;
          $cast(t, mon_txn.clone());
```

Con la variable en *scoreboard* podemos usarla de etiqueta para saber que instrucciones fallan, para ello creamos una variable *string* en *scoreboard* que sea la etiqueta de error más la variable que hará de etiqueta de instrucción, a su vez enviamos esta nueva variable en un comando de *uvm_error* cuando la comparación falle, de esta forma en el resumen final de las notificaciones de UVM saldrá el nombre de todas las instrucciones que hayan fallado.

```
tag = {"SCOREBOARD_FAIL_",actual_txn.instru};// string instruccion
        `uvm_error(tag, msg)
```

3.4.2 Auto eliminado de instrucciones

Para realizar esta propuesta se realizó mediante *flags*, hay dos tipos *flag* de error que salta cuando ocurre un error y *flag* de instrucción que indica en que instrucción ocurrió el error.

Para crear estas variables no hace falta tener en cuenta los tiempos por lo que se podrá en *unico_item* de forma *static* para que se comparta entre todas las instancias de *item*. Se llaman *error* y *errorlist*.

Para facilitar la construcción del algoritmo se usará una variable *enum* que le dará a cada instrucción un número, se llama *check*. Este *enum* junto con el resto de las variables se pueden ver en las variables vistas de *unico_item*.

El algoritmo funciona de la siguiente forma:

1. Se elige la instrucción que se va a comprobar.
2. Comprueba que esa instrucción no ha sido eliminada o si ha saltado algún error anterior.
3. Arranca la simulación y comprueba si no habido un error durante la simulación.
4. Ejecuta la instrucción y para la simulación.
5. Repite desde el paso 2 hasta que termine el *loop*, si es de doble instrucción repite desde el paso 1.
6. Terminado el *loop* vuelve a empezar desde 1.

En el momento que se detecte error la secuencia terminar lo antes posible y volverá a empezar reiniciando el *flag* de error, si detecta que la instrucción ha sido eliminada pasara a la siguiente.

```
  task body();
    do begin
      req_item_h = unico_item::type_id::create("req_item_h");
      req_item_h.error = 0;
      start_item(req_item_h);
        $signal_force("/top/rst","0",,,250,);
      finish_item(req_item_h);

      req_item_h.check = req_item_h._lw;
      repeat(33) begin
        if(!req_item_h.errorlist[req_item_h.check] && !req_item_h.error)begin
          start_item(req_item_h);
          if(!req_item_h.error)begin
            req_item_h.constraint_type("lw");
            assert(req_item_h.randomize());
            req_item_h.idata[11:7]=i;
            i++;
            `uvm_info({get_type_name(),":body"},{"Sending transaction
",req_item_h.convert2string()}, UVM_MEDIUM)
          end
          finish_item(req_item_h);
        end
      end

      req_item_h.check = req_item_h._addi;
      repeat(num_repetionsI_S_SB) begin
        if(!req_item_h.errorlist[req_item_h.check] && !req_item_h.error)begin
          start_item(req_item_h);
          if(!req_item_h.error)begin
            req_item_h.constraint_type("addi");
            assert(req_item_h.randomize());
            `uvm_info({get_type_name(),":body"},{"Sending transaction
",req_item_h.convert2string()}, UVM_MEDIUM)
          end
          finish_item(req_item_h);
        end
      end
```

Para detectar que instrucción ha dado error se aprovecha la etiqueta de instrucción en el *scoreboard* mediante un case dependiendo de que instrucción estaba cuando dio error se levanta el *flag* correspondiente.

```
actual_txn.error = 1;

        case(actual_txn.instru)

          "addi":  actual_txn.errorlist[0] = 1;

          "slti":  actual_txn.errorlist[1] = 1;

          "sltiu": actual_txn.errorlist[2] = 1;

          "andi":  actual_txn.errorlist[3] = 1;

          "ori":   actual_txn.errorlist[4] = 1;
```

```
"xori":  actual_txn.errorlist[5] = 1;
"lui":   actual_txn.errorlist[6] = 1;
"auipc": actual_txn.errorlist[7] = 1;
"add":   actual_txn.errorlist[8] = 1;
"slt":   actual_txn.errorlist[9] = 1;
"sltu":  actual_txn.errorlist[10] = 1;
"and":   actual_txn.errorlist[11] = 1;
"or":    actual_txn.errorlist[12] = 1;
"xor":   actual_txn.errorlist[13] = 1;
"sub":   actual_txn.errorlist[14] = 1;
"beq":   actual_txn.errorlist[15] = 1;
"bne":   actual_txn.errorlist[16] = 1;
"lw":    actual_txn.errorlist[17] = 1;
"sw":    actual_txn.errorlist[18] = 1;
"slli":  actual_txn.errorlist[19] = 1;
"srli":  actual_txn.errorlist[20] = 1;
"srai":  actual_txn.errorlist[21] = 1;
"sll":   actual_txn.errorlist[22] = 1;
"srl":   actual_txn.errorlist[23] = 1;
"sra":   actual_txn.errorlist[24] = 1;
"jal":   actual_txn.errorlist[25] = 1;
"jalr":  actual_txn.errorlist[26] = 1;
"blt":   actual_txn.errorlist[27] = 1;
"bltu":  actual_txn.errorlist[28] = 1;
"bge":   actual_txn.errorlist[29] = 1;
"bgeu":  actual_txn.errorlist[30] = 1;
default: actual_txn.errorlist = 0;
```

Una cosa a tener en cuenta es que cada vez que la secuencia vuelve a empezar por que ha detectado un error es necesario hacer un *reset* en los registros, para ello se usó un comando especifico de "**QuestaSim**" que permite forzar durante el tiempo que quieres un valor y después lo devuelve a como estaba.

Ese comando es *$signal_force* y se vio en el código del algoritmo, sus argumentos son muchos, pero solo interesan 3, el primero la dirección de la señal, el segundo el valor que deseas y el quinto el tiempo en nanosegundos que deseas que este activo el comando.

3.5 Errores y soluciones

1. Revisando el diseño que sería el *gold-model* se descubrió un error en la realización de las instrucciones de desplazamiento, era más notorio en las no inmediatas. El error era porque a la hora de realizar el desplazamiento cogía todos los bits para el cálculo y solo puede desplazar hasta 31. Solución: se limitó en la ALU a 5 bits la cantidad que coge para el cálculo del desplazamiento con y sin signo.

2. Cuando se hicieron las primeras pruebas tras poner el *gold_model* los resultados eran incoherentes. Era porque al ser *gate_level* tenía un tiempo de *hold* el cual no se tuvo en cuenta. Solución: Se aumentó el tiempo del reloj para compensar.

3. El PC del *gold_level* tardaba unos ciclos en actuar tras el *reset* comparado con el DUT. El error estaba en que después del *reset* las entradas tardaban un tiempo en recuperarse de la meta-estabilidad lo que provocaba un retraso en el PC. Solución: Se cambiaron todas las variables por bits en la interfaz y las que se pudo en el *gold_level*.

4. Aunque esto no es un error en sí mismo se mencionará aquí porque al final se descartó. Había varias secuencias con sus secuencias virtuales que actuaban una después de otra en *start_test*. Para añadir varias secuencias después de usar la plantilla de secuencia **SEQUENCE_only** junto la plantilla de secuencia virtual **VSEQUENCE_only** se tiene que incluir tanto en *sequences_pkg* y *virtual_sequences_pkg*, después se debe de instanciar la nueva secuencia virtual y activar la función de inicio en la función *run_phase* de *start_test*. Aquí un ejemplo de un experimento que se hizo con el multiplicador.

```
virtual task run_phase(uvm_phase phase);

    vsequence1 vseq1 = vsequence1::type_id::create("vseq1");
    vsequence2 vseq2 = vsequence2::type_id::create("vseq2");

    phase.raise_objection(this);
    `uvm_info({get_type_name(),":run"}, "Starting test...", UVM_LOW)

    //Connect sequence to sequencer and start
    init_vseq(vseq1);  // Using method from test base class to assign sequence
handles
    vseq1.start(null); // null because no target sequencer
    init_vseq(vseq2);

    vseq2.start(null);
```

Capítulo 4. Verificación del Banco de Pruebas

4.1 Introducción

En este apartado veremos los resultados de simular y que aspecto tiene.

Se verá desde dos puntos de vista uno en el que se verificará un DUT que esta todo perfectamente y no da errores, después, otro DUT el cual genera errores en algunas instrucciones.

4.2 Verificación con modelo sin errores

Figura 11. Consola de "QuestaSim" librerías

Se puede ver como carga las librerías sin problemas gracias a la compilación de las librerías realizada antes y de cómo en el comando *vsim* pide de cargarlas.

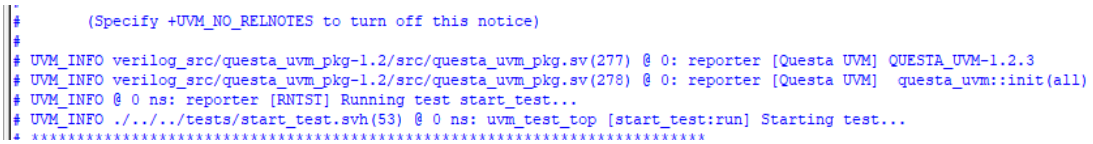


Figura 12. Consola de "QuestaSim" Inicio

Al observar las primeras notificaciones de UVM significa que la simulación ya está preparada para arrancar. Con un run *all* la simulación comienza.

Durante la simulación aparecen en cada iteración 5 notificaciones de diferentes partes que va indicando el viaje que hacen las variables.

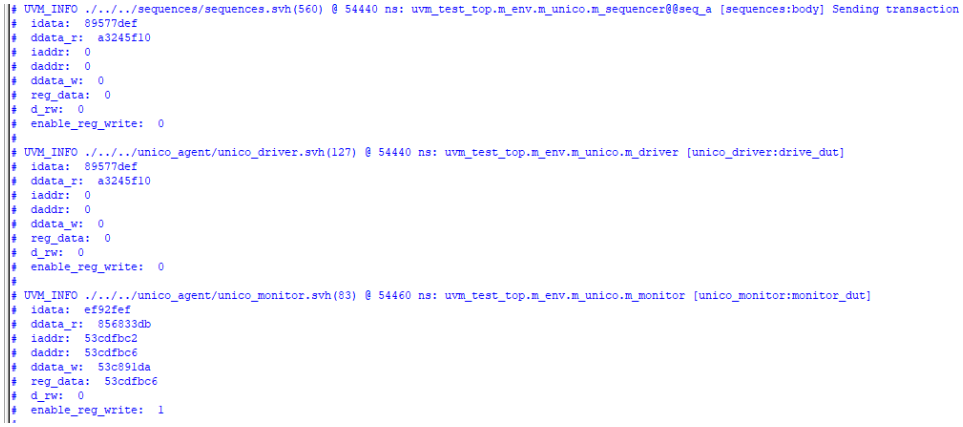


Figura 13. Consola de "QuestaSim" Notificaciones 1ª parte

La primera notificación es de la secuencia indica que valores ha generado el *random*.

La segunda del drive, indica que valores introdujo en las FIFOs de entrada. Normalmente estas dos primeras notificaciones tienen los mismos valores.

La tercera notificación es del monitor e indica los valores leídos en la DUT por el monitor. Se puede apreciar que los valores son diferentes a las anteriores, eso se debe a que hay un pequeño retraso en las primeras interacciones provocando que en las FIFOs haya por lo menos un valor, así que en la siguiente interacción se verá en monitor el valor de drive de la anterior.

```
# UVM_INFO ./../../environment/scoreboard.svh(141) @ 54460 ns: uvm_test_top.m_env.m_scoreboard [SCOREBOARD] INPUT:
#  idata:  ef92fef
#  ddata_r:  856833db
#  iaddr:  53cdfbc2
#  daddr:  53d724b0
#  ddata_w:  53c891da
#  reg_data:  53cdfbc6
#  d_rw:  0
#  enable_reg_write:  1
#
# UVM_INFO ./../../environment/scoreboard.svh(83) @ 54460 ns: uvm_test_top.m_env.m_scoreboard [SCOREBOARD_PASS] Expected RSP: {
#  idata:  ef92fef
#  ddata_r:  856833db
#  iaddr:  53cdfbc2
#  daddr:  53d724b0
#  ddata_w:  53c891da
#  reg_data:  53cdfbc6
#  d_rw:  0
#  enable_reg_write:  1
# }  Actual RSP: {
#  idata:  ef92fef
#  ddata_r:  856833db
#  iaddr:  53cdfbc2
#  daddr:  53cdfbc6
#  ddata_w:  53c891da
#  reg_data:  53cdfbc6
#  d_rw:  0
#  enable_reg_write:  1
# }
```

Figura 14. Consola "QuestaSim" Notificaciones 2ª parte

La cuarta notificación es el valor de las variables del modelo de referencia captadas por el *scoreboard*. Podemos ver que los valores son iguales que la notificación anterior, eso indica que en la comparación no habrá error.

La última notificación es de la comparación del *scoreboard*, si esta no da ningún problema es solo una notificación más, pero si da error entonces se convertirá en una notificación de error y se pondrá en marcha toda la maquinaria de errores.

```
# UVM_INFO ./../../tests/start_test.svh(65) @ 54542 ns: uvm_test_top [start_test:run] ... test completed
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 54542 ns: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO ./../../unico_agent/unico_driver.svh(66) @ 54542 ns: uvm_test_top.m_env.m_unico.m_driver [unico_driver:report] 1355 sequence items
# Coverage: covered = 1594, total = 33554673 (85.71%)
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 6777
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [SCOREBOARD]  1354
# [SCOREBOARD_PASS]  1354
# [TEST_DONE]     1
# [sequences:body]  1353
# [start_test:run]     2
# [unico_driver:drive_dut]  1355
# [unico_driver:report]     1
# [unico_monitor:monitor_dut]  1354
# ** Note: $finish    : C:/questasim64_2020.2/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
```

Figura 15. Consola de "QuestaSim" resumen

Cuando la simulación termina indica el estado de los *coverages* y muestra el resumen, aquí se indica todos los tipos de notificaciones con sus etiquetas, como se puede ver no ha habido ninguna notificación de error.

4.3 Verificación con modelo con errores

```
# UVM_INFO ./../../unico_agent/unico_monitor.svh(83) @ 16380 ns: uvm_test_top.m_env.m_unico.m_monitor [unico_monitor:monitor_dut]
#  idata:  5104a63
#  ddata_r:  b658cc69
#  iaddr:  242c
#  daddr:  0
#  ddata_w:  4d2
#  reg_data:  0
#  d_rw:  0
#  enable_reg_write:  0
#
# UVM_INFO ./../../environment/scoreboard.svh(142) @ 16380 ns: uvm_test_top.m_env.m_scoreboard [SCOREBOARD] INPUT:
#  idata:  5104a63
#  ddata_r:  b658cc69
#  iaddr:  1dc4
#  daddr:  1
#  ddata_w:  4d2
#  reg_data:  1
#  d_rw:  0
#  enable_reg_write:  0
#
# UVM_ERROR ./../../environment/scoreboard.svh(91) @ 16380 ns: uvm_test_top.m_env.m_scoreboard [SCOREBOARD_FAIL_blt] Expected RSP: {
#  idata:  5104a63
#  ddata_r:  b658cc69
#  iaddr:  1dc4
#  daddr:  1
#  ddata_w:  4d2
#  reg_data:  1
#  d_rw:  0
#  enable_reg_write:  0
# }  Actual RSP: {
#  idata:  5104a63
#  ddata_r:  b658cc69
#  iaddr:  242c
#  daddr:  0
#  ddata_w:  4d2
#  reg_data:  0
#  d_rw:  0
#  enable_reg_write:  0
# }
```

Figura 16. Consola de "QuestaSim" Notificaciones con error

Aquí se puede apreciar el error generado por una instrucción, como los valores del monitor y del *scoreboard* son diferentes termina dando un error en la comparación, ahora justo después es posible que haga una ejecución más antes de que la secuencia se termine y reinicie, esto ocurre porque el error salta justo después del *if*, aunque no pasa siempre. Se puede ver, a diferencia de cuando la comparación era correcta, la instrucción que se estaba ejecutando en ese momento, que en el caso del ejemplo es *blt*.

```
# UVM_INFO ./../../unico_agent/unico_driver.svh(66) @ 348782 ns: uvm_test_top.m_env.m_unico.m_driver [unico_driver:report] 8603 sequence items
# Coverage: covered = 8818, total = 33554673 (85.72%)
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :43036
# UVM_WARNING :    0
# UVM_ERROR :   18
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [SCOREBOARD]  8620
# [SCOREBOARD_FAIL_bge]     2
# [SCOREBOARD_FAIL_bgeu]     2
# [SCOREBOARD_FAIL_blt]     2
# [SCOREBOARD_FAIL_bltu]     2
# [SCOREBOARD_FAIL_jal]     2
# [SCOREBOARD_FAIL_jalr]     2
# [SCOREBOARD_FAIL_sll]     1
# [SCOREBOARD_FAIL_slli]     1
# [SCOREBOARD_FAIL_sra]     1
# [SCOREBOARD_FAIL_srai]     1
# [SCOREBOARD_FAIL_srl]     1
# [SCOREBOARD_FAIL_srli]     1
# [SCOREBOARD_PASS]  8602
# [TEST_DONE]     1
# [sequences:body]  8584
# [start_test:run]     2
# [unico_driver:drive_dut]  8603
# [unico_driver:report]     1
# [unico_monitor:monitor_dut]  8620
# ** Note: $finish    : C:/questasim64_2020.2/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
```

Figura 17. Consola de "QuestaSim"

En el resumen se ven todas las instrucciones que dieron errores, y como algunas se ejecutaron dos veces y otras no. Se ven un total de 12 instrucciones por lo que la secuencia se debió repetir unas 13 veces.

4.4 Comparación de resultados

La diferencia más obvia entre ellos es que el que tiene errores tarda más ya que tiene que reiniciar cada vez que falla alguna instrucción, podemos ver como uno ha realizado 1353 ejecuciones de *random* mientras que el otro son 8584, más de 6 veces de uno respecto al otro. Por otro lado, aunque ha realizado muchas más interacciones el *covergroup* no parece haber subido mucho.

Capítulo 5. Conclusiones

5.1 Conclusión

El resultado final del proyecto es bueno, de hecho, la verificación sin errores si tuvo errores que encontró el banco de pruebas, donde el DUT tenía mal puestas las fun3 de dos instrucciones, esto indica que el objetivo de principal del proyecto se llegó a completar.

Usando esta forma de proceder se podría comprobar mediante *gold_model* cualquier otro modulo no solo RISC_V.

EL objetivo opcional que aún está pendiente de completar es el de crear una guía para la utilización del proyecto por otros alumnos, ya que requiere trasladar el proyecto a otro ordenador y los proyectos están muy ligados al directorio de donde están, sin haber ningún método de exportación de proyectos como tal, para poder hacer una guía en condiciones será necesaria investigar todos los arreglos de los directorios necesarios para que funcione lo que supone un periodo largo de investigación, como el resultado que salga de esta investigación puede ser demasiado para una “pequeña” guía, se pone sobre la mesa la posibilidad de hacer un video explicativo para realizar la guía. Todo esto tendría que realizar post TFG como trabajo extra para facilitar el acceso a próximos alumnos

5.2 Mirada al futuro

Este es el comienzo para empezar hacer más verificaciones en UVM, y con proyecto como este permitir a alumnos conocer esta arquitectura de verificación.

También expandir este proyecto a más RISC_V, como la realización con RISC_V segmentado con el *mono_cyclo* de *gold_model*. Proyecto bastante complicado ya que en diferentes ciclos de reloj los resultados se van mostrando en el segmentado por lo que la comprobación también deberá ser segmentada y simultanea con otras instrucciones.

Capítulo 6. Bibliografía

[1] Rincón de SystemVerilog ”Verificación en UVM”
https://dsd.webs.upv.es [Online]

[2] ChipVerify, “SystemVerilog y UVM tutorials”
https://www.chipverify.com [Online]

[3] Herraiz, M. *Diseño de un banco de pruebas con UVM (Universal verfication methodology)*. ETSIT., Curso 2018-19

Capítulo 7. Anexo

7.1 Lista de acrónimos

DUT	Device Under Test
EDA	Electronic Design Automation
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
IC	Integrated Circuit
IP	Intellectual Property
OVM	Open Verification Methodology
RHS	Right Hand Side
RTL	Register-Transfer Level
SIP	Semiconductores IP
SV	SystemVerilog
TLM	Transaction-Level Modeling
UVM	Universal Verification Methodology
VHDL	VHSIC Hardware Description Language
VMM	Verification Methodology Manual