 (*)

Vulture Platform Computer (VPC)

Principles of Operation: Specification & Instruction Set

Revision v.2.2 12/13/2017 – Final Term (patched with novel feature)

By Rafael Brito
rab405@g.harvard.edu
rafaebrito@gmail.com

CSCI E-93
Computer Architecture – Fall 2017
Prof James Frankel

HARVARD Extension School

Table of Contents

Document Revision of Changes

Version	Date	Author	Description
1.0	10/01/2017	Rafael Brito rab405@g.harvard.edu	Created first version of document. Graded 133/150
1.1	10/14/2017	Rafael Brito rab405@g.harvard.edu	Added LoadIR on block diagram; Removed references of "delay slot" since there is no pipelining and corrected the $ra; Removed reference on "int" as pseudo instruction; Created a separate section for novel feature (in progress).
1.2	10/29/2017	Rafael Brito rab405@g.harvard.edu	Instructions are parsed via blank spaces; Changed the name of multiple instructions; Load/Store word instructions changed syntax and names; SIBEQ, SIBNEQ, SRJAL moved to jump instructions. Added three jump instructions: JR, JEQ and JNEQ. Added LW, SW and LR instructions for memory. total register decimal numbers. Multiple placeholders to fill.
1.3	11/05/2017	Rafael Brito rab405@g.harvard.edu	Added instruction RXNOR
1.4	11/12/2017	Rafael Brito rab405@g.harvard.edu	Remove any "LI" references. Adding "LR" and "WSR" under R-type
1.4a	11/12/2017	Rafael Brito rab405@g.harvard.edu	LW gets address of memory; added hex example on SIBEQ; added labels .ascii and .address directives
1.5	11/13/2017	Rafael Brito rab405@g.harvard.edu	ALU OpCode is now 0x08 (instead of 0x00); Changed again LW/SW: they Load/Store the word plus get the memory address.
2.0	12/03/2017	Rafael Brito rab405@g.harvard.edu	Added section on Finite State Machines of VPC – both CPU and Memory; Added the diagnostic section with reference of switched, LEDs

			and buttons. Updated the block diagram with MemMux
2.1	12/12/2017	Rafael Brito rab405@g.harvard.edu	Officially deprecated SRPC, SILOAD, SISTORE instructions; Updated FSM section: VPC has been implemented with a single FSM; Updated Block diagram with correct memory length and remove deprecated instructions and components.
2.2	12/13/2017	Rafael Brito rab405@g.harvard.edu	Added barrel shifter (instructions SBSRL SBSLL) as a novel feature, deprecating the intended hardware interruption.

(*) Animal from cover is oreilly™ - http://shop.oreilly.com/product/9780596007072.do

Introduction

The purpose of this document is to define the principles of operation, specifications and instructions of Vulture Platform Computer, referred on this documents as "VPC".

Specifications and Block Design

VPC is a very simple 32-bit instruction size machine created for academic purposes inspired on RISC architecture, more specific on MIPS technology (MIPS)

VPC has 32 general-purpose registers accessible for the programmer:

Register Type/Name	Qty	Address (5-bit binary)	Address (Decimal)	Purpose/Description
$z0	1	00000	0	Read-only (constant) register for "zero" operations
$a0-$a1	2	00001-00010	1-2	Argument registers
$g0-$g7	8	00011-01010	3-10	General-Purpose Registers
$sp	1	01101	11	Stack pointer (not currently used - reserved for future use)
$fp	1	01100	12	Frame pointer (not currently used - reserved for future use)
$r0-$r7	8	01111-10100	13-20	Return memory address Registers
$t0-$t3	4	10101-11000	21-24	Temporary Registers (not persistent)
$s0-$s4	5	11001-11101	25-29	Saved registers –used for returning values from routines
$gp	1	11110	30	Global Pointer (not currently used - reserved for future use)
$ra	1	11111	31	Return Address

VPC has two other registers that are "protected" from the programmer does not have access:

- PC (Program Counter) (16-bit)
- IR (Instruction Register)

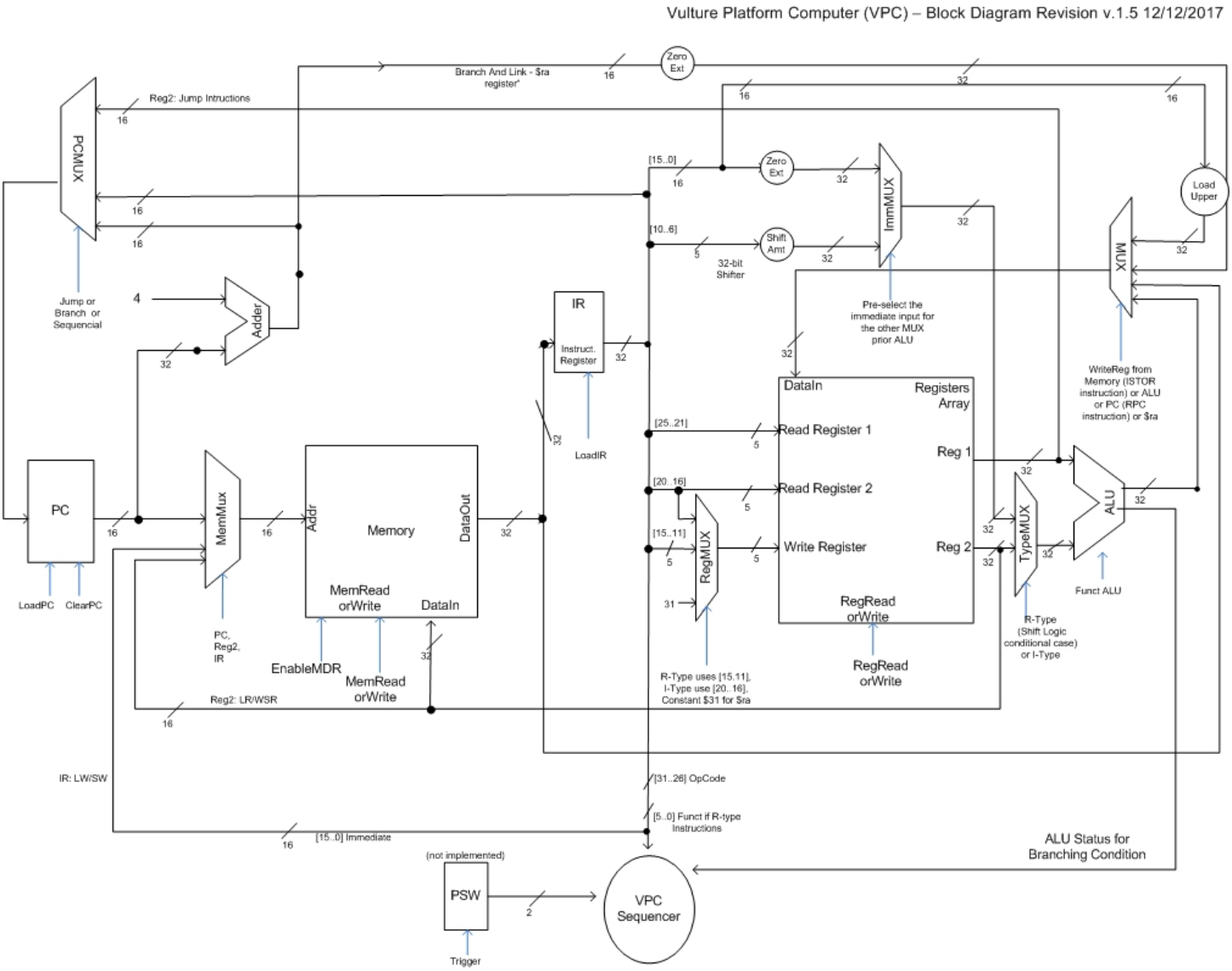
There was planned a third protected register, PSW, aimed for the novel VPC feature: hardware interrupt mechanism for I/O devices. For time constraints, barrel shifter was implemented as the novel feature

For simplification and academic deployment, memory addressing and arithmetic operations (ADD/SUB) were implemented based 16-bit. For those operations, the immediate field and lower order of registers are used.

Memory operations are always double word.

All instructions on numerical operations (I-types) are with signed integers. Floating points and unsigned integers are not supported.

Block diagram

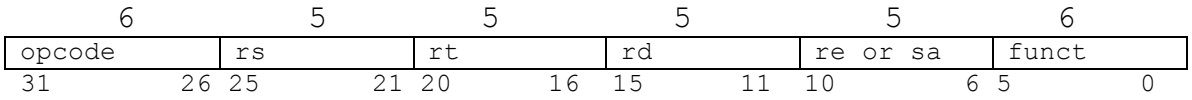


Instructions Set Architecture (ISA)

VPC instruction set has 6 bits for the Operation Code (opcode) with two types basic instruction formats: Register-type (usually instructions that start with the letter “R” and relies on multiple registers), Immediate-type (instructions rely on a immediate value). Instructions naming convention have is instructions start with letter “R” (R-type) or “I” (I-type).

The access of the stack is through the registers $sp, $fp, $gp and $ra and they do have special instructions to manage them - will use the regular R-type and I-type instructions.

Register-type Instructions (R-Type)



Register Destination (rd) = register that will have the final result
Register Source (rs and rt and re) = registers will be source of the instructions
Shift amount (sa) = used for Shift Amount instructions
Operation Code (opcode) = Instruction identifier
Function (funct) = opcode modifier code (ALU instructions)

Short List of R-type instructions

Instruction	Short Description
RADD	Addition between two GPRs
RSUB	Subtraction between two GPRs
RAND	AND between two GPRs
ROR	OR between two GPRs
RNOR	NOR between two GPRs
RXOR	XOR between two GPRs
RXNOR	XNOR between two GPRs
RSLT	Set on less than (compare two GPRs)
SRSRL	Shift right logical
SRSLL	Shift left logical
LR	Load Word from memory address on register
WSR	Store word of $rd to the memory address on register
SBSRL	Barrel Shift right logical
SBSLL	Barrel Shift left logical

RADD

Bits	6	5	5	5	5	6
Fields	0x08 SPECIAL	rs	rt	rd	"sbz"	0x01
Format	RADD $rd $rs $rt					
Description	"Addition between two GPRs" The content of register rt is added to the content of the register rs and storage on register rd. The values are signed integers and there is no overflow protection. If overflow protection occurs, register rd does not get overwritten.					
Operation	GPR[rd] ← GPR[rs] + GPR[rt]					
Example	RADD $g0 $g0 $g1 In this example, general-purpose register $g0 value is being added by the value on the $g1 register.					
Example In binary	001000 00011 00100 00011 00000 000001					
Example In hex	0x00641801					

RSUB

Bits	6	5	5	5	5	6
Fields	0x08 SPECIAL	rs	rt	rd	"sbz"	0x02
Format	RSUB $rd $rs $rt					
Description	"Subtraction between two GPRs" The content of register rs is subtracted by the content of the register rt and storage on register rd. The values are signed integers and there is no overflow protection. If overflow protection occurs, register rd does not get overwritten.					
Operation	GPR[rd] ← GPR[rs] - GPR[rt]					
Example	RSUB $g0 $g0 $g1 In this example, general-purpose register $g0 value is being subtracted by the value on the $g1 register.					
Example In binary	001000 00011 00100 00011 00000 000010					
Example In hex	0x00641802					

RAND

Bits	6	5	5	5	5	6
Fields	0x08 SPECIAL	rs	rt	rd	"sbz"	0x03
Format	RAND $rd $rs $rt					
Description	"AND between two GPRs" Bitwise two registers $rs and $bt and stores on $rd. Used to zero registers.					
Operation	GPR[rd] ← GPR[rs] & GPR[rt]					
Example	RAND $g0 $g0 $z0 In this example, general-purpose register $g0 is being zeroed.					
Example In binary	001000 00011 00000 00011 00000 000011					
Example In hex	0x00601803					

ROR

Bits	6	5	5	5	5	6
Fields	0x08 SPECIAL	rs	rt	rd	"sbz"	0x04
Format	ROR $rd $rs $rt					
Description	"OR between two GPRs" Bitwise two registers $rs OR $rt and stores on $rd.					
Operation	GPR[rd] ← GPR[rs] | GPR[rt]					
Example	ROR $g0 $a0 $a1 In this example, general-purpose register $g0 is the result of the "OR" between $a0 and $a1.					
Example In binary	001000 00001 00010 00011 00000 000100					
Example In hex	0x20221804					

RNOR

Bits	6	5	5	5	5	6
Fields	0x08 SPECIAL	rs	rt	rd	"sbz"	0x05
Format	RNOR $rd $rs $rt					
Description	"NOR between two GPRs" Bitwise two registers $rs NOR $rt and stores on $rd.					
Operation	GPR[rd] ← ~(GPR[rs] | GPR[rt])					
Example	RNOR $g0 $g1 $g2					

	In this example, general-purpose register $g0 is the result of the "NOR" between $g1 and $g2.
Example In binary	001000 00100 00101 00011 00000 000101
Example In hex	0x00851805

RXOR

Bits	6	5	5	5	5	6
Fields	0x08 SPECIAL	rs	rt	rd	"sbz"	0x06
Format	RXOR $rd $rs $rt					
Description	"XOR between two GPRs" Bitwise two registers $rs Exclusive-OR $rt and stores on $rd.					
Operation	GPR[rd] ← GPR[rs] ^ GPR[rt]					
Example	RXOR $g0, $g1, $g2 In this example, general-purpose register $g0 is the result of the "XOR" between $g1 and $g2.					
Example In binary	001000 00100 00101 00011 00000 000110					
Example In hex	0x00851806					

RXNOR

Bits	6	5	5	5	5	6
Fields	0x08 SPECIAL	rs	rt	rd	"sbz"	0x0A
Format	RXNOR $rd $rs $rt					
Description	"XNOR between two GPRs" Bitwise two registers $rs Exclusive-NOR $rt and stores on $rd.					
Operation	GPR[rd] ← GPR[rs] ^ GPR[rt]					
Example	RXNOR $g0 $g1 $g2 In this example, general-purpose register $g0 is the result of the "XNOR" between $g1 and $g2.					
Example In binary						
Example In hex	- VPC					

RSLT

Bits	6	5	5	5	5	6
Fields	0x08 SPECIAL	rs	rt	rd	"sbz"	0x07
Format	RSLT $rd $rs $rt					
Description	"Set on less than (compare two GPRs)" If $rs is less than $rt the register $rd gets "1"; otherwise $rd gets "0" . Numbers are signed integers.					
Operation	GPR[rd] ← (GPR[rs] < GPR[rt])? 1 : 0					
Example	RSLT $g0 $g1 $g2 Where $g1 is -2 and $g2 is 2 In this example, general-purpose register $g0 gets "1"					
Example In binary	001000 00100 00101 00011 00000 000111					
Example In hex	0x00851807					

SRSRL

Bits	6	5	5	5	5	6
Fields	0x08 SPECIAL	rs	"sbz"	rd	sa	0x08
Format	SRSRL $rd $rs sa					
Description	"Shift Right Logic" Register $rs 's value is shifted right by sa bits, inserting zeros into the high order bits. The result of the operation is set on $rd register.					
Operation	GPR[rd] ← 0^{sa} || $GPR[rs]_{31..sa}$					
Example	SRSRL $t1 $t1 16 Where $t1 is 0x00A00000 In this example, general-purpose register $t0 gets 0x00000002					
Example In binary	001000 10110 00000 10110 10000 001000					
Example In hex	0x22C0B409					

SRSLL

Bits	6	5	5	5	5	6
Fields	0x08	rs	"sbz"	rd	sa	0x09

	SPECIAL					
Format	SRSLL $rd $rs sa					
Description	"Shift Left Logic" Register $rs 's value is shifted left by sa bits, inserting zeros into the low order bits. The result of the operation is set on $rd register.					
Operation	GPR[rd] ← GPR[rs]$_{31..sa}$||0^{sa}					
Example	SRSLL $g0 $g1 4 Where $g1 is 0xFFFF In this example, general-purpose register $g0 gets 0xFFF0					
Example In binary	001000 10110 00000 10110 10000 001001					
Example In hex	0x22C0B409					

SBSRL

Bits	6	5	5	5	5	6
Fields	0x08 SPECIAL	rs	"sbz"	rd	sa	0xB
Format	SBSRL $rd $rs sa					
Description	"Barrel Shift Right Logic" Register $rs 's value is shifted right by sa bits, rotating bits into the high order bits. The result of the operation is set on $rd register.					
Operation	GPR[rd] ←GPR[rs]$^{sa..0}$ || GPR[rs]$_{31..sa}$					
Example	SBSRL $t1 $t1 16					
Example In binary						
Example In hex						

SBSLL

Bits	6	5	5	5	5	6
Fields	0x08 SPECIAL	rs	"sbz"	rd	sa	0xC
Format	SBSLL $rd $rs sa					
Description	"Barrel Shift Left Logic" Register $rs 's value is shifted left by sa bits, rotating bits into the low order bits. The result of the operation is set on $rd register.					
Operation	GPR[rd] ← GPR[rs]$_{31..sa}$|| GPR[rs]$^{sa..0}$					
Example	SBSLL $g0 $g1 4					

Example In binary	
Example In hex	

LR

Bits	6	5	5	16
Fields	0x17	$rs	rd	"sbz"
Format	LR $rd $rs			
Description	"Load Word from memory address on register $rs into register $rd"			
Operation	T: vAddr ← $rs mem ← LoadMemory(WORD, vAddr) GPR[rt] is undefined T+1: GPR[rd] ← mem			
Example	LR $s3 $a0			
Example In binary				
Example In hex	0x			

WSR

Bits	6	5	5	16
Fields	0x18	$rs	rd	"sbz"
Format	WSR $rd $rs			
Description	"Store word of $rd to the memory address that register $rs contains"			
Operation	T: vAddr ← $rs mem ← StoreMemory(GPR[rd], vAddr)			
Example	WSR $s3 $a0			
Example In binary				
Example In hex	0x			

Immediate-type Instructions (I-Type)

6	5	5	16
opcode	rs	rd	immediate

```
31      26 25       21   20      16  15                              0
```

Operation Code (opcode) = Instruction identifier
Register Source (rs and rt) = registers will be source of the instructions

Instruction	Short Description
IADD	Addition between one GPR and a signed extended number
ISUB	Subtraction between one GPR and a signed extended number
IAND	AND between one GPR and a zero extended number
IOR	OR between one GPR and a zero extended number
INOR	NOR between one GPR and a zero extended number
IXOR	XOR between one GPR and a zero extended number
ISLT	Set on less than (compare a GPR a signed extended number)
SILU	Load an upper number on a GPR
LW	Load Word based on a memory address (label)
SW	Store Word based on a memory address (label)

IADD

Bits	6	5	5	16
Fields	0x01	rs	rd	immediate
Format	`IADD $rd $rs immediate`			
Description	"Addition between one GPR and a signed extended number" The content of register rs is added by the immediate value. The values are signed integers and there is no overflow protection. If overflow protection occurs, register rd does not get overwritten. If the result is smaller of a word size, the instructions will automatically zero-extend it.			
Operation	GPR[rd] ← GPR[rs] + SigExtImm			
Example	`IADD $g0 $g0 1500` In this example, general purpose register $g0 value is being added by the 1500 (decimal).			
Example In binary	`000001 00011 00011 0000010111011100`			
Example In hex	`0x046305DC`			

ISUB

Bits	6	5	5	16
Fields	0x02	rs	rd	immediate
Format	`ISUB $rd $rs immediate`			
Description	"Subtraction between one GPR and a signed extended number" The content of register rs is subtracted by the immediate value. The			

	values are signed integers and there is no overflow protection. If overflow protection occurs, register rd does not get overwritten. If the result is smaller of a word size, the instructions will automatically zero-extend it.
Operation	GPR[rd] ← GPR[rs] - SigExtImm
Example	`ISUB  $g0  $g0  1500` In this example, general purpose register $g0 value is being added by the 1500 (decimal).
Example In binary	`000010 00011 00011 0000010111011100`
Example In hex	`0x086305DC`

IAND

Bits	6	5	5	16
Fields	0x03	rs	rd	immediate
Format	`IAND  $rd  $rs  immediate`			
Description	Binary			
Operation	GPR[rd] ← GPR[rs] & ZeroExtImm			
Example	`IAND  $g0  $g1  0xFA` $g0 receives the result of the AND between $g1 with 0xFA			
Example In binary	`000011 00100 00011 0000000011111010`			
Example In hex	`0x0C6300FA`			

IOR

Bits	6	5	5	16
Fields	0x04	rd	rs	immediate
Format	`IOR  $rd  $rs  immediate`			
Description	“OR between one GPR and a zero extended number” Bitwise “OR” a register and an immediate value and stores the result in a register			
Operation	GPR[rd] ← GPR[rs] | ZeroExtImm			
Example	`IOR  $g0  $g1  0xFA` $g0 receives the result of the “OR” between $g1 with 0xFA			
Example In binary	`000100 00100 00011 0000000011111010`			
Example	`0x106300FA`			

In hex	

INOR

Bits	6	5	5	16
Fields	0x05	rs	rd	immediate
Format	INOR $rd $rs immediate			
Description	"NOR between one GPR and a zero extended number" Bitwise "NOR" a register and an immediate value and stores the result in a register			
Operation	GPR[rd] ← ~(GPR[rs] | ZeroExtImm)			
Example	INOR $g0 $g1 0xFA $g0 receives the result of the "NOR" between $g1 with 0xFA			
Example In binary	000101 00100 00011 0000000011111010			
Example In hex	0x146300FA			

IXOR

Bits	6	5	5	16
Fields	0x06	rs	rd	immediate
Format	IXOR $rd $rs immediate			
Description	"XOR between one GPR and a zero extended number" Bitwise "XOR" a register and an immediate value and stores the result in a register			
Operation	GPR[rd] ← GPR[rs] ^ ZeroExtImm			
Example	IXOR $g0 $g1 0xFA $g0 receives the result of the "XOR" between $g1 with 0xFA			
Example In binary	000110 00100 00011 0000000011111010			
Example In hex	0x186300FA			

ISLT

Bits	6	5	5	16
Fields	0x07	rs	rd	immediate
Format	ISLT $rd $rs immediate			
Description	"Set on less than (compare a GPR a signed extended number)" If $rs is less than $rt the register $rd gets "1"; otherwise $rd gets "0" .			

	Numbers are signed integers.
Operation	GPR[rd] ← (GPR[rs] < SigExtImm)? 1 : 0
Example	`ISLT  $g0  $g1  -300` Where $g1 is -2 In this example, general-purpose register $g0 gets "0"
Example In binary	`000111 00100 00011 1111111011010100`
Example In hex	`0x1C83FED4`

SILU

Bits	6	5	5	16
Fields	0x10	"sbz"	rd	immediate
Format	`SILU  $rd immediate`			
Description	"Load an upper number on a GPR" The 16-bit immediate is shifted left 16 bits (signed extended value); the low order 16 bits are set to zeros; result is stored in general register rd. This is used in conjunction with IADD to load large numbers into the registers.			
Operation	GPR[rd] ← (immediate || 0^{16})			
Example	`SILU  $a0  0x00FF` $a0 gets the value 0x00FF0000.			
Example In binary	`00100 0000 0000 0001 0000 0000 1111 1111`			
Example In hex	`0x400100FF`			

SILOAD

Bits	6	5	5	16
Fields	0x11	Base	rd	offset
Format	`SILOAD  $rd  base offset`			
Description	"Load word on a GPR based on an offset" [deprecated – NOT IMPLEMENTED - see LW and LR] The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register rt.. In this case "base" is the register $sp in VPC.			
Operation	T: vAddr ← ((offset15)16 || offset15..0) + GPR[base]			

	mem ← LoadMemory(WORD, vAddr) GPR[rt] is undefined T+1: GPR[rd] ← m00
Example	`SILOAD  $g0  $sp -8` $g0 loads the word from the $sp memory address minus 8
Example In binary	`010001 01101 00011 1111111111111000`
Example In hex	`0x45A3FFF8`

SISTORE

Bits	6	5	5	16
Fields	0x12	base	rt	offset
Format	`SISTORE  $rt base offset`			
Description	"Store Word on a GPR based on an offset" [deprecated – NOT IMPLEMENTED - see SW and WSR] The 16-bit offset is sign-extended and added to the contents of general register base to form a virtual address. The contents of the word on register $rt is stored at the memory location specified by the effective address. In this case "base" is the register $sp in VPC.			
Operation	T: vAddr ← ((offset15)16 || offset15..0) + GPR[base] mem ← StoreMemory(GPR[rt], vAddr)			
Example	`SISTORE  $g0 $sp -4` $sp with an offset of minus 4 (memory address) gets the word from $g0			
Example In binary	`010010 01101 00011 1111111111111100`			
Example In hex	`0x49A3FFFC`			

LW

Bits	6	5	5	16
Fields	0x15	$rt	rd	immediate
Format	`LW $rd $rt label`			
Description	"Load Word" copy word address at source RAM location to the register $rd The source RAM location is referred by the label. Register $rd get the actual word from the address. Register $rt gets the address			
Operation	T: vAddr ← immediate mem ← LoadMemory(WORD, vAddr) T+1:			

	GPR[rd] ← mem GPR[rt] ← vAddress

Example	`LW $t2 $g0 MYIOADDR`
	
	Where IOADDR is `0x00FF00`
	As per directive on assembler file
	`IOADDR: .address "0x00FF00"`
	
	$t2 gets the word on that address
	$g0 gets `"0x00FF00"`

Example In binary	
Example In hex	

SW

Bits	6	5	5	16
Fields	0x16	rt	rs	immediate
Format	`SW $rs rt label`			
Description	"Store Word" copy word (4 bytes) from register source to RAM location. The RAM location is referred by the label. Register $rt gets the address.			
Operation	T: vAddr ← immediate GPR[rt] ← vAddress mem ← StoreMemory(GPR[rs], vAddr)			

Example	`SW $r3 $g0 MYIOADDR`
	
	Where MYIOADDR is `0x00FF00`
	
	As per directive on assembler file
	
	`MYIOADDR: .address "0x00FF00"`
	
	$g0 gets the address of MYIOADDR
	The content of $r3 is stored on MYIOADDR address.			

Example In binary				
Example In hex				

Jump Type Instructions

Instruction	Short Description
SIBEQ	Branch (and link) to a target if equal
SIBNEQ	Branch (and link) to a target not if equal
SRJAL	Jump unconditionally to the immediate value (label)
JR	Jump unconditionally to the register
JEQ	Jump if equal
JNEQ	Jump if not equal

SIBEQ

Bits	6	5	5	16
Fields	0x13	rs	rt	target
Format	SIBEQ $rs $rt target			
Description	"Branch (and link) if equal" A target address is generated during compiler time from the label of the assembler code. Then register rs and rt are compared. If the two registers are equal, then the program storages $ra as PC+4 (for return address) and branches to the target address, with a delay of one instruction.			

Operation	T:
	condition ← (GPR[rs] = GPR[rt])
	T+1:
	if condition then
	$ra ← PC + 4
	PC ← target
	endif			

Example	SIBEQ $z0 $z0 main Label is converted to an address at the compiling time. In this case "main" points to decimal 4 PC will jump to address at label if $rs and $rt are equal.			
Example In binary	010011 00000 00000 0000000000000100			
Example In hex	0x4C000004			

SIBNEQ

Bits	6	5	5	16
Fields	0x14	rs	rt	target
Format	SIBNEQ $rs $rt target			

Description	"Branch (and link) not if equal" A target address is generated during compiler time from the label of the assembler code. Then register rs and rt are compared. If the two registers are not equal, then the program storages $ra as PC+4 (for return address) and branches to the target address, with a delay of one instruction.

Operation	T:
	condition ← (GPR[rs] <> GPR[rt])
	T+1:
	if condition then
	$ra ← PC + 4
	PC ← target
	endif

Example	`SIBNEQ  $g0  $g1 display` Label is converted to an address at the compiling time. In this case "display" points to decimal 4 PC will jump to address at label if $rs and $rt are NOT equal.
Example In binary	
Example In hex	

SRJAL

Bits	6	5	5	16
Fields	0x0A	rs	rd	immediate
Format	`SRJAL  $rd  $rs immediate`			
Description	Jump to the immediate (generated at compiler time from label) and link Program will jump unconditionally to address on immediate and link with a delay of one instruction. The address of the instruction is placed in general register $rs and $rd. Register are redundant rs and rd.			
Operation	immediate ← GPR[rs] GPR[rd] ← PC + 4 GPR[rs] ← PC + 4 GPR[$ra] ← PC + 4 T+1: PC ← immediate			
Example	`SRJAL  $g0  $t0 main`			

	Program jumps to address on $g0 and $t0 gets the return address. OBS: At programming time, this instruction will take label as a 3rd parameter and the compiler will add the instruction address on register.
Example In binary	`001010 00011 00000 0000000000100000`
Example In hex	`0x`

JR

Bits	6	5	5	16
Fields	0x0B	"sbz"	rd	"sbz"
Format	`JR $rd`			
Description	Jump immediate to address on register rd without link.			
Operation	PC ← GPR[rd]			
Example	`JR $ra` Program jumps to address on $ra			
Example In binary	`00101100000111100000000000000000`			
Example In hex	`0x2C1E0000`			

JEQ

Bits	6	5	5	5	11
Fields	0x15	rs	rt	rd	"sbz"
Format	`JEQ $rs $rt $rd`				
Description	"Jump if equal" rs and rt are compared. If the two registers are equal, then the program jump to $rd register without link.				

Operation	T:
	condition ← (GPR[rs] = GPR[rt])
	T+1:
	if condition then
	PC ← $rd
	endif				

Example	`JEQ $a0 $g0 $ra`				

Example In binary	
Example In hex	

JNEQ

Bits	6	5	5	5	11
Fields	0x0D	rs	rt	rd	"sbz"
Format	JNEQ $rs $rt $rd				
Description	"Jump if equal" rs and rt are compared. If the two registers are not equal, then the program jump to $rd register without link.				

Operation	T:
	condition ← (GPR[rs] <> GPR[rt])
	T+1:
	if condition then
	PC ← $rd
	endif				

Example	JNEQ $a0 $g0 $ra				
Example In binary					
Example In hex					

Other types of instructions: Pseudo Instructions

At this time, there is no pseudo instruction but designer acknowledges that this might change.
There is no specific instructions to rotate / logical shifts of one or multiple bit(s).

VPC Compiler Directives

Comments

They are "#" or " --". They can be a separate line or on inline with the code.

Labels

"main" and other user-defined

The VPC compiler accepts the labels for the instructions jump and link and branch and link.
On the VPC assembler program, the labels are single word followed by a ":".
Every program is required at least the "**main**" label for the program.
The compiler looks for incorrect cross-references labels and duplication of labels.
On both cases, the compilation of the program fails.
There is no specific order for labels. The program start with assembler line "00" pointing to the first instruction of the label "main".

".ascii" and ".address" label directives

The instructions SRJAL, SIBEQ, SIBNEQ, LW and SW are immediate instructions but at compiler time they get as a parameter a label.
The label is defined as the following

```
message1: .ascii "Enter First Number "
message2: .ascii "Enter Second Number "
message3: .ascii "Result of Multiplication is "
inputmemory1: .address "0x0440"
inputmemory2: .address "0x0480"

LW $a0 message1 -- $a0 will get the address of the message1 label
```

Program Stack and Data Stack

The compiler defines the size of the program stack and data stack, based on number of instructions.
The data stack address starts 5 words after program stack and it can use the directive ".ascii" for labels.

VPC Novel Feature: Interruptions

At this time, there is a fourth protected register, PSW, planned for the novel VPC feature of hardware interrupts mechanism.

This feature is in planning phase and data path not yet depicted on the block diagram.

The proposal is PSW being activated by a trigger signal and sending 2-bit to the sequencer.

The sequencer completes the current instruction and then suspends the program execution pushing registers $g0-$g7, $r0-$r7, $t0-$t3, $ra on the stack (more details to be defined).

The sequencer then will set PC to a specific instruction value and execute a routine to load registers $s0-$s3 with words from a specific memory address. The sequencer stores registers $g0-$g7, $r0-$r7, $t0-$t3, $ra from the stack and resumes the program execution.

More details will be provided in the future releases of the document.

VPC Finite State Machines

Preamble

VPC will use the following clocking directives:

- On the rising edge of the clock, VPC FSM will transition from state to state;
- On the high clock (1), the signals will be propagated thru the gates;
- On the falling edge of the clock, all destructions actions are taken, such as changing VPC registers
- On the low clock (1), the VPC FSM computes the next state;
- Contrary from stated on previous version of this document (v2.0), there will be **one** FSM

State	Description	Actions / Transition
Initial	VPC Machine started or reset	PC ← "0" All registers are zero'd NextState ← FetchAwaitMemoryInstruction

FetchAwaitMemoryInstruction	Assign Memory Inputs and Prepare to read instruction	If mem_dataready_inv =1
		NextState ← FetchReadingInstruction
		Else
		Hold State
		End if

FetchReadingInstruction	Read word from memory based on address from the PC.	If mem_dataready_inv = 0
		NextState ←Fetched
		Else
		Hold State
		End if

	Signal the memory and await for the done signal	
ParseInstruction	Parse the instruction word, getting opcodes, registers, functs, etc. Next state depends on on opcode and setup memory mux appropriately.	If UserData MemIO Write NextState ← WriteBackAwaitMemory Else UserData MemIO Read NextState ← FetchAwaitMemoryUserData Else NextState ← ExecuteInstruction
ExecuteInstruction	Initial Execution State, just to add one more cycle for signal replication	NextState ← ExecuteInstruction2
ExecuteInstruction2	Latch registers based on opcodes	PC is updated NextState ← FetchAwaitMemoryInstruction

WriteBackAwaitMemory	Write back to RegisterFile or Setup memory Write	mem_rw ← 1
		mem_data_write ← register
		
		If mem_dataready_inv =0
		Hold State
		Else
		NextState ← WriteBackWriting
		End if

WriteBackWriting	Signal the memory and await the done signal	mem_addressready ← 0
		If mem_dataready_inv = 1
		NextState ←Written
		Else
		Hold State
		End if

Written	Confirmed data was written in memory; Execute the Instruction (write executions only updates the registers with	mem_addressready ← 0 NextState ← ExecuteInstruction

	memory addresses)	

FetchAwaitMemoryUserData	Reads data from memory address given by instruction	If mem_dataready_inv =1
		NextState ←FetchReading
		Else
		Hold State
		End if

FetchReadingUserData	Signal the memory and await for the done signal	If mem_dataready_inv = 0
		NextState ←Fetched
		Else
		Hold State
		End if

FetchedUserData	Data is stable and passed as an internal register to instruction Execution	NextState ← ExecuteInstruction

FPGA Diagnostics

The following are the switches and buttons on VPC FPGA (Diagnostics purposes):

Switches

SW0	Clock Hold
SW1	Memory Suspend (not tested)
SW2	IR Suspend (not tested)
SW3	
SW4	mem_addr on Segment
SW5	IR on Segment
SW6	PC on Segment
SW11..7	VPC Registers 0 to 31 on Segment
SW17..12	**Throttle**

Red LEDs – FSM Indicators

LEDR0	Initial
LEDR1	FetchAwaitMemoryInstruction
LEDR2	FetchReadingInstruction
LEDR3	ParseInstruction
LEDR4	ExecuteInstruction
LEDR5	ExecuteInstruction2

LEDR6	
LEDR7	WriteBackAwaitMemory
LEDR8	WriteBackWriting
LEDR9	Written
LEDR10	FetchAwaitMemoryUserData
LEDR11	FetchReadingUserData
LEDR12	FetchedUserData
LEDR13	
LEDR14	
LEDR15	
LEDR16	
LEDR17	

Green LEDs – variables

LEDG0	mem_dataready_inv
LEDG1	mem_addressready
LEDG2	mem_rw
LEDG3	InstructionExecuted
LEDG4	Memmux_enable
LEDG5	
LEDG6	
LEDG7	

Keys

KEY0	cpu_reset (not implemented)
KEY1	clock_step
KEY2	mem_reset
KEY3	