RISC: Documentation

Nikhil Shah (17CS10030)
Amatya Sharma (17CS30042)

Register Conventions:

1. Decided registers and the register usage convention.

Register Name	Register Number	Register Code	Function
$zero	0	00000	Zero
$flags	1	00001	4 of its beginning bits indicate 4 different flags: Z, carry, sign and overflow
$sp	2	00010	Stack pointer
$res1-$res2	3 - 4	00011 - 00100	Reserved registers
$ra	5	00101	Return address pointer
$v0-$v1	6 - 7	00110 - 00111	Return values
$a0 - $a3	10 - 11	01001 - 01011	arguments
$t0 -$t6	12 - 18	01100 - 10010	temporaries
$hi	19	10011	Most significant bits
$lo	20	10100	Least Significant bits
$t7 -$t10	21 - 24	10101 - 11000	temporaries
$s0 -$s7	25 - 31	11001 - 11111	Saved temporaries

Operation Codes:

2. Design a suitable instruction format and instruction encoding. While deciding the opcode, you should keep provisions for adding more instructions to the ISA.

Opcode	Class	Functions
000	Arithmetic	add,multu,mult,comp,addi,compi
001	Logic	and,xor
010	Shift	shll,shrl,shllv,shrlv,shra,shrav
011	Memory	lw,sw
100	Branch	b,br,bz,bnz,bcy,bncy,bs,bnv,call,ret

Arithmetic : 000

Function	Usage	Binary Representation
Add	add rs,rt	0000
Multiply (Unsigned)	multu rs,rt	0001
Multiply (Signed)	mult rs,r	0010
Complement	comp rs,rt	0011
Add immediate	addi rs,imm	0100
Complement immediate	compi rs,imm	0101

Logic : 001

Function	Usage	Binary Representation

AND	and rs, rt	0000
XOR	xor rs, rt	0001

Shift : 010

Function	Usage	Binary Representation
Shift left Logical	shll rs, sh	0000
Shift right Logical	shrl rs, sh	0001
Shift left Logical variable	shllv rs, rt	0010
Shift right Logical variable	shrlv rs, rt	0011
Shift right Arithmetic	shra rs, sh	0100
Shift right Arithmetic variable	shrav rs, rt	0101

Memory:011

Function	**Usage**	**Binary Representation**
Load Word	lw rt, imm(rs)	0000
Store Word	sw rt, imm(rs)	0001

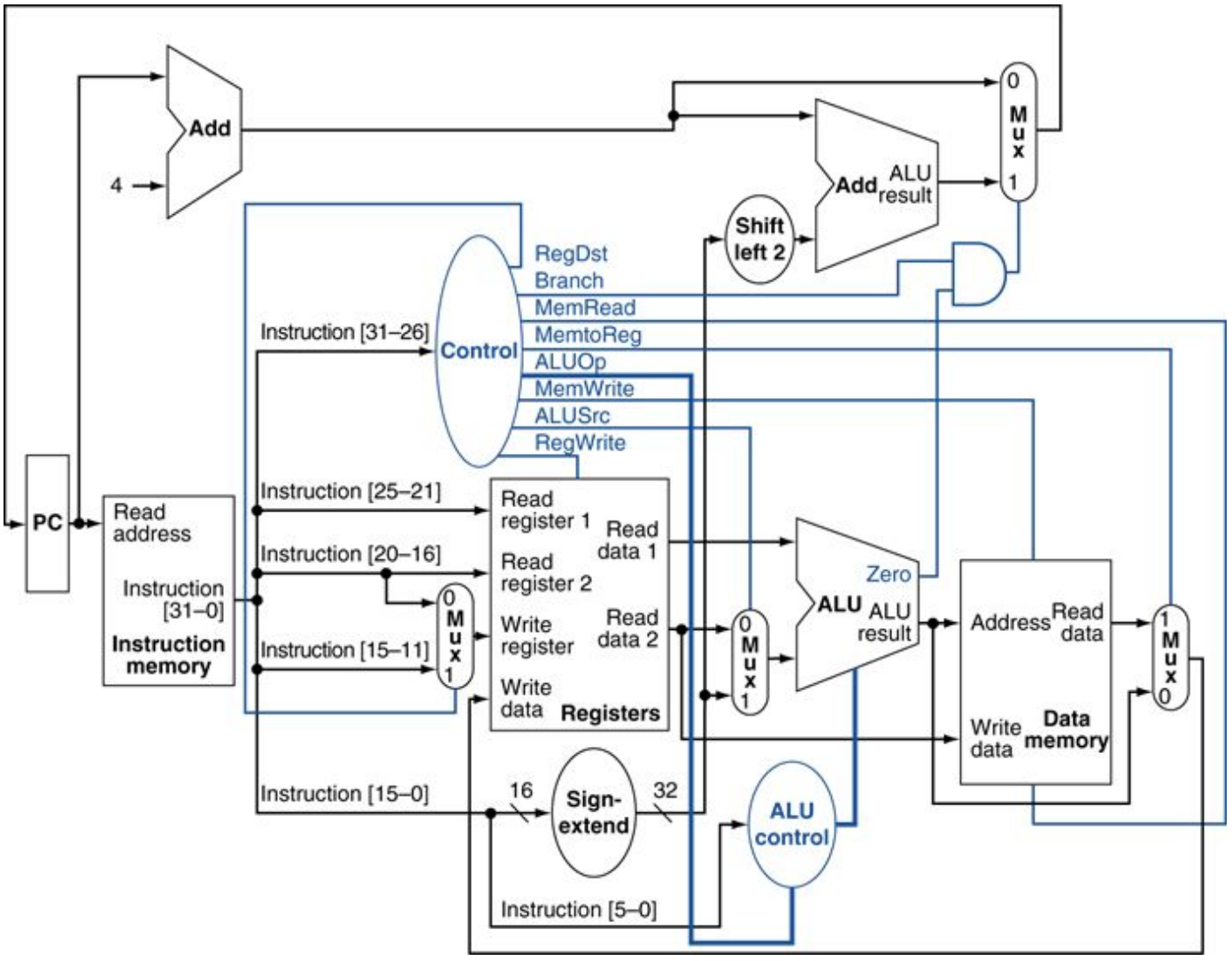
Branch :100

Function	Usage	Binary Representation
Unconditional branch	b L	0000
Branch Register	br rs	0001
Branch on zero	bz L	0010

Branch on not zero	bnz L	0011
Branch on Carry	bcy L	0100
Branch on No Carry	bncy L	0101
Branch on Sign	bs L	0110
Branch on Not Sign	bns L	0111
Branch on Overflow	bv L	1000
Branch on No Overflow	bnv L	1001
Call	Call L	1010
Return	Ret	1011

Architecture:

RISC



Modules:

1. Instruction Fetch:

1.1. PC Incrementer: Increments the next program counter to either the jump value (L or ra) or adds one to the current program counter. Works synchronously with the clock.

1.2. Instruction Memory: Loads the instructions at the current program counter from the BRAM instantiated for instructions only.

2. **Instruction Decoder:** Segments the instruction into interpreteble register addresse, jump values, immediate addresses, operation code and immediate values/shift amounts.
3. **Register Bank:** Stores 32 registers each of 32 bits . Writes (on negedge) in values of the input registers on write enable signal. Always outputs values of registers read from instructions.
4. **Control:** Depending on the operation it activates various modules/hardwares with different functionalities (all flags defined later).
5. **Input Decider:** Decides between immediate and register values depending on the instruction type.
6. **ALU:** Main arithmetic and logical unit of the processor, employs hierarchical design incorporating hybrid adders and array multipliers for both signed and unsigned operations.
7. **Data Memory:** Higher level module for BRAM instantiated for data memory.
8. **Branch Logic:** Depending on the type of the branch instruction, returns the jump value (direct address of the next instruction).
9. **Write Back:** Chooses between ALU output and memory output to write back to destination register.

Flags/Controls:

1.**regDst** —> use when you need three registers to check, which register to write to branch,

2.**memRead**

3.**memToReg** --> decide b/w alu ou/p and data memory o/p

4.**memWrite** —> select line to data memory

5.**aluSrc** –> differentiates between register and constant for ALU
6.**regWrite** –> write to reg or not (if write : synchronous)
7.**regBranch** –> to decide whether jump to (rs) or L
8.**raWrite** –> to decide if it call is used , write back to ra register
9.**isMult** –> to write back hi and lo

Assumptions and Other Points To Be Noted

1. Since it is not mentioned in the assignment the mode of addressing to be used, we have used **direct addressing** for branch instructions.
2. Block RAMs have a peculiar issue that they have significant delay in fetching data (around 0.5 clock cycles) from the RAM. Hence, we have divided the input clock into two parts, a slower one and a faster one. The faster one is eight times faster and is used to fetch data from Block RAM whereas the slower one is used for other modules.
3. Block RAMs have addresses as 0, 1, 2, …. Hence, we have used **PC+1 instead of PC+4**.

Appendix

Note :
1. Register file : Don’t use high level code,
Use 32 * 32 one bit input mux’s .
Don’t compare entire 32 bit register addresses.

Synthesis Statistics

Adders/Subtractors - 7

- **a. 32-bit adder - 5**
- **b. 32-bit adder carry out - 2**

Registers - 78

- **a. 1-bit register - 17**
- **b. 32-bit register - 61**

Latches - 14

- **a. 1-bit latch - 11**
- **b. 4-bit latch - 3**

Multiplexers - 7

- **a. 32-bit 16-to-1 multiplexer - 3**
- **b. 32-bit 32-to-1 multiplexer - 4**

Logic shifters - 5

a. **32-bit shifter logical left - 3**
b. **32-bit shifter logical right - 2**

Xors - 3

a. **1-bit xor4 - 2**
b. **32-bit xor2 - 1**

Implementation:

- ➔ Logic core generator:
- ➔ Depth of mem = Not known prior (put bound accordingly)
- ➔ 2 ports : i/p (din [31:0]) , o/p(dout[31:0])
- ➔ i/p: clk (may not be the same as system clk)
- ➔ writeEnable signal (to read or write)
  - ◆ Write operation : o/p will be copied to dout register
  - ◆ Read operation : whatever is read is o/p to dout (for decoding purposes)
- ➔ Create block ram for instruction memory /*Assume already done*/
- ➔ Initialise Inst Memory using file with extension '.coe'
  - ◆ Contains content fetched from inst read from inst memory
  - ◆ 2 keywords should always be present
    - Memory_initialisation_radix /* contains values with given radix , for our case : '2;'*/
    - Memory_initialisation_vector /* actual contents of the file , for our case : -didn't tell- */
    - /* instructions are separated with ',/*space*/ ' and the whole code ends with a ';' */