Alexandria University
Faculty of Engineering
Computers and Communications Department
Spring Final Exam, June 2017

جامعة الإسكندرية
كلية الهندسة
قسم هندسة الحاسبات والاتصالات
امتحان نهاية الفصل الدراسي الثاني (يونيو ٢٠١٧)

Course Title and Code Number:
Computer Architecture (CC 322)
Time Allowed: 2 hours

اسم المقرر والرقم الكودي له:
معماريات الحاسب (CC 322)
الزمن: ساعتين

Attempt All Questions: (45 marks)

Question 1 (20 marks)

a) Modify the single-cycle MIPS processor shown in Figure 1 to implement each of the following instructions. Indicate the changes required to the datapath, control unit, and control signals shown by Table 1

i. `sb` ii. `mult, mflo, mfhi` ii. `jalr`

b) Modify the multicycle MIPS processor shown in Figure 2 to implement each of the following instructions. Indicate the changes required to the datapath, control unit, and control FSM shown by Figure 3. Describe any other changes that are required.

i. `addi` ii. `sllv` ii. `lwinc`

The `lwinc` instruction (load with post increment instruction, which updates the index register to point to the next memory word after completing the load) is equivalent to:

```
lw $rt, imm($rs)
addi $rs, $rs, 4
```

c) Can you modify the Single-cycle and Multicycle processors to implement the following R-type instructions without altering the processor state elements? If your answer is yes, indicate the needed modifications to both the datapath and control units. If your answer is no, indicate why, and show how these instructions can be performed using existing MIPS instructions.

i. `beq rs, rt, rd #if reg(rs)==reg(rt) then PC=reg(rd) else PC=PC=4;`

ii. `addm rd, rs, rt # reg(rd)=reg(rs)+Mem(rt)`

d) Redesign the MIPS Multicycle datapath and controller to use a single read/write port register file.

Question 2 (15 marks)

a) The following program is running on a 5-stage (F, D, E, M, W) pipelined MIPS processor:

```
      add $s0, $0, $0    # i = 0
      add $s1, $0, $0    # sum = 0
      addi $t0, $0, 10   # $t0 = 10
loop:
      slt $t1, $s0, $t0  # if (i < 10), $t1 = 1, else $t1 = 0
      beq $t1, $0, done  # if $t1 == 0 (i > = 10), branch to done
      add $s1, $s1, $s0  # sum = sum + i
      addi $s0, $s0, 1   # increment i
      j loop
done:
```

i. Calculate the CPI of this code segment on a single-cycle and multi-cycles MIPS processor.
ii. Calculate the CPI of this code segment on a pipelined MIPS processor without a hazard unit. Assume `nop` instructions are inserted to resolve RAW hazards.
iii. Calculate the CPI of this code segment on a pipelined MIPS processor having a full (data and control) hazard unit with forwarding and stalling mechanisms. Show timing of one loop cycle as follows:

Instruction	Clock Cycle Number														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`slt $t1, $s0, $t0`	F	D	E	M	W										
`beq $t1, $0, done`															
`add $s1, $s1, $s0`															
`addi $s0, $s0, 1`															
`j loop`															

b) Explain how to extend the pipelined MIPS processor to handle the `addi` and `j` instructions. Give particular attention to how the pipeline is flushed when a jump takes place.

c) A standard benchmark consists of approximately 20% loads, 10% stores, 15% branches, 5% jumps, and 50% R-type instructions. Assume that 30% of the loads are immediately followed by an instruction that uses the result, requiring a stall, and that 25% of the branches are mispredicted, requiring a flush. Assume that jumps always flush the subsequent instruction.

i. Compute the average CPI of the single-cycle, multicycle, and pipelined processors.
ii. Compare the execution time for a program with 10 billion instructions on the three processors. The delay of various circuit elements is shown the following table.
iii. Indicate only the most important parameter that needs optimization from both the fabrication technology and computer architecture point of views to improve the overall performance of each processor.

Element	Parameter	Delay (ps)
register clk-to-Q	t_{pcq}	30
register setup	t_{setup}	20
multiplexer	t_{mux}	25
ALU	t_{ALU}	300
memory read	t_{mem}	250
register file read	t_{RFread}	150
register file setup	$t_{RFsetup}$	20
equality comparator	t_{eq}	40
AND gate	t_{AND}	15
register file write	$t_{RFwrite}$	100
memory write	$t_{memwrite}$	220

<u>Question 3</u> (15 marks)

a) An 8-word cache has the following parameters: b, block size given in numbers of words; S, number of sets; N, number of ways; and A, number of address bits. Consider the following repeating sequence of `lw` addresses (given in hexadecimal):

74 A0 78 38C AC 84 88 8C 7C 34 38 13C 388 18C

Assuming least recently used (LRU) replacement for associative caches, determine the effective miss rate if the sequence is input to the following caches, ignore compulsory misses.

i. direct mapped cache, b = 1 word
ii. fully associative cache, b = 2 words
iii. two-way set associative cache, b = 2 words
iv. direct mapped cache, b = 4 words

b) Consider a cache with the following parameters:
N (associativity) = 4, b (block size) = 16 words, W (word size) = 32 bits, C (cache size) = 32 K words, A (address size) = 32 bits. You need to consider only word addresses.

i. Show the tag, set, block offset, and byte offset bits of the address. State how many bits are needed for each field?
ii. What is the size of all the cache tags in bits?
iii. Suppose each cache block also has a valid bit (V) and a dirty bit (D). What is the size of each cache set, including data, tag, and status bits?
iv. Design the cache using the building blocks in the following figure and a small number of two-input logic gates. The cache design must include tag storage, data storage, address comparison, data output selection, and any other parts you feel are relevant. Note that the multiplexer and comparator blocks may be any size (n- or p-bit wide, respectively), but the SRAM blocks must be 8K × 8 bits. Be sure to include a neatly labeled block diagram. You need only design the cache for reads.

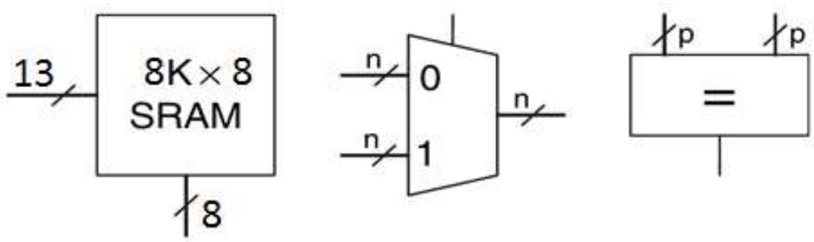

Figure: Cache Building blocks

c) Consider a virtual memory system that can address a total of 2^{32} bytes. You have unlimited hard drive space, but are limited to 4 GB of semiconductor (physical) memory. Assume that virtual and physical pages are each 8 KB in size.

i. How many bits is the physical and virtual addresses?
ii. What is the number of physical and virtual pages in this system?
iii. How many bits are the virtual and physical page numbers?
iv. How many page table entries will the page table contain?
v. How many bytes long is each page table entry taking the valid bit into consideration?
vi. Sketch the layout of the page table. What is the total size of the page table in bytes?

Good Luck,

Examiner: Dr. Mohammed Morsy

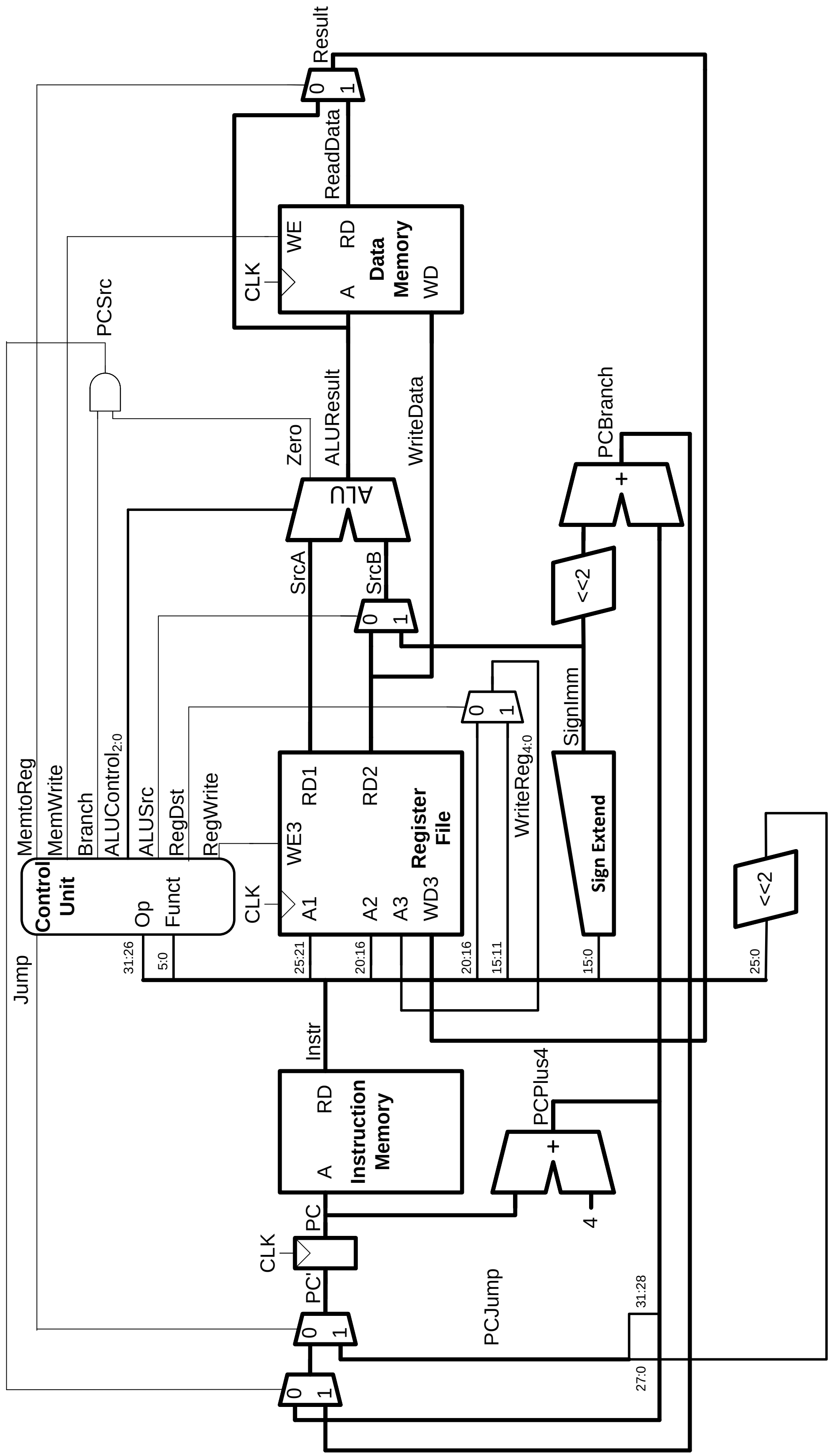

Figure 1: Single-cycle MIPS processor

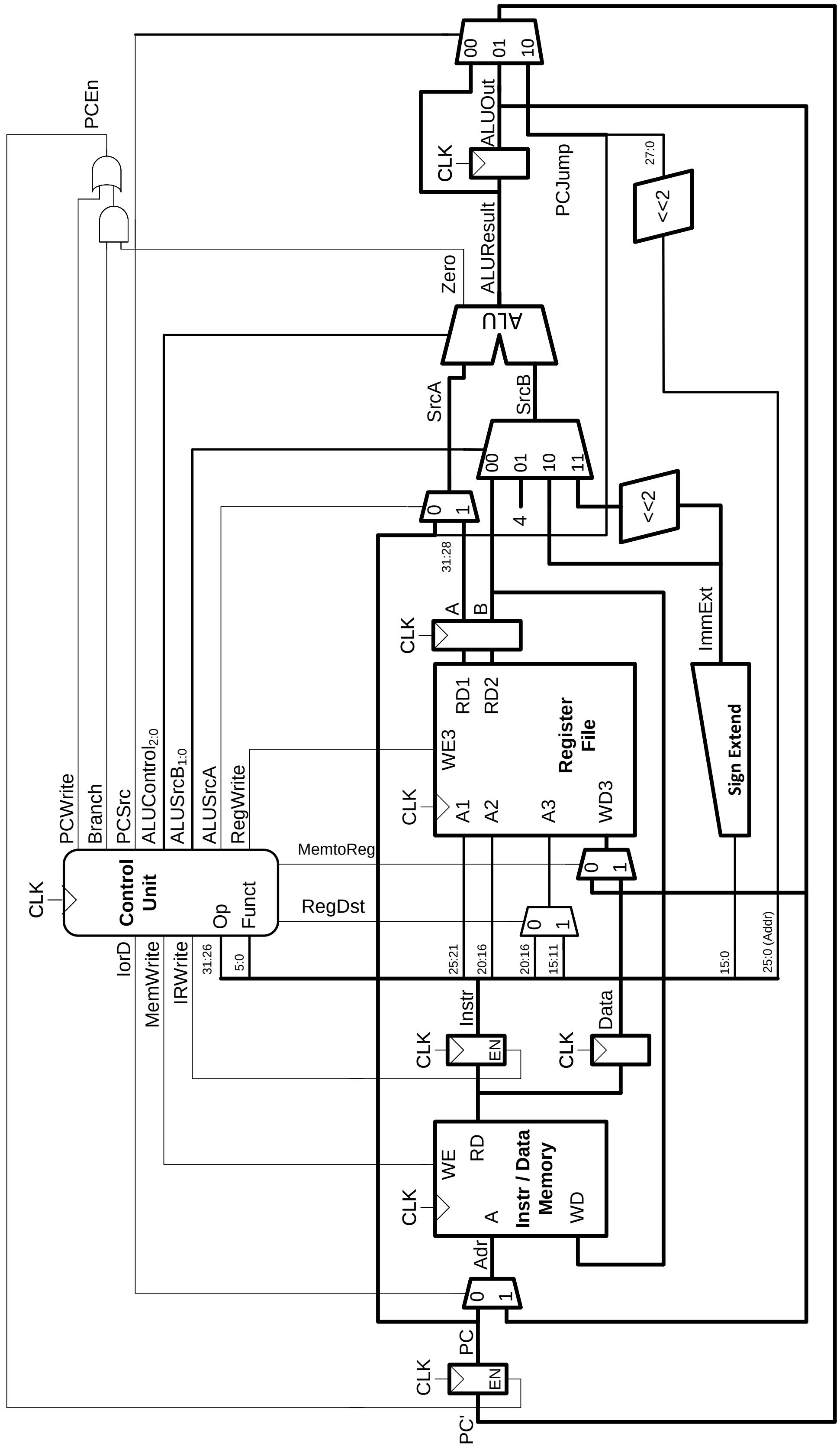

Figure 2: Multicycle MIPS processor

Table 1: Single cycle MIPS processor control signals

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump			
`R-type`	000000	1	1	0	0	0	0	10	0			
`lw`	100011	1	0	1	0	0	1	00	0			
`sw`	101011	0	X	1	0	1	X	00	0			
`beq`	000100	0	X	0	1	0	X	01	0			
`j`	000100	0	X	X	X	0	X	XX	1			

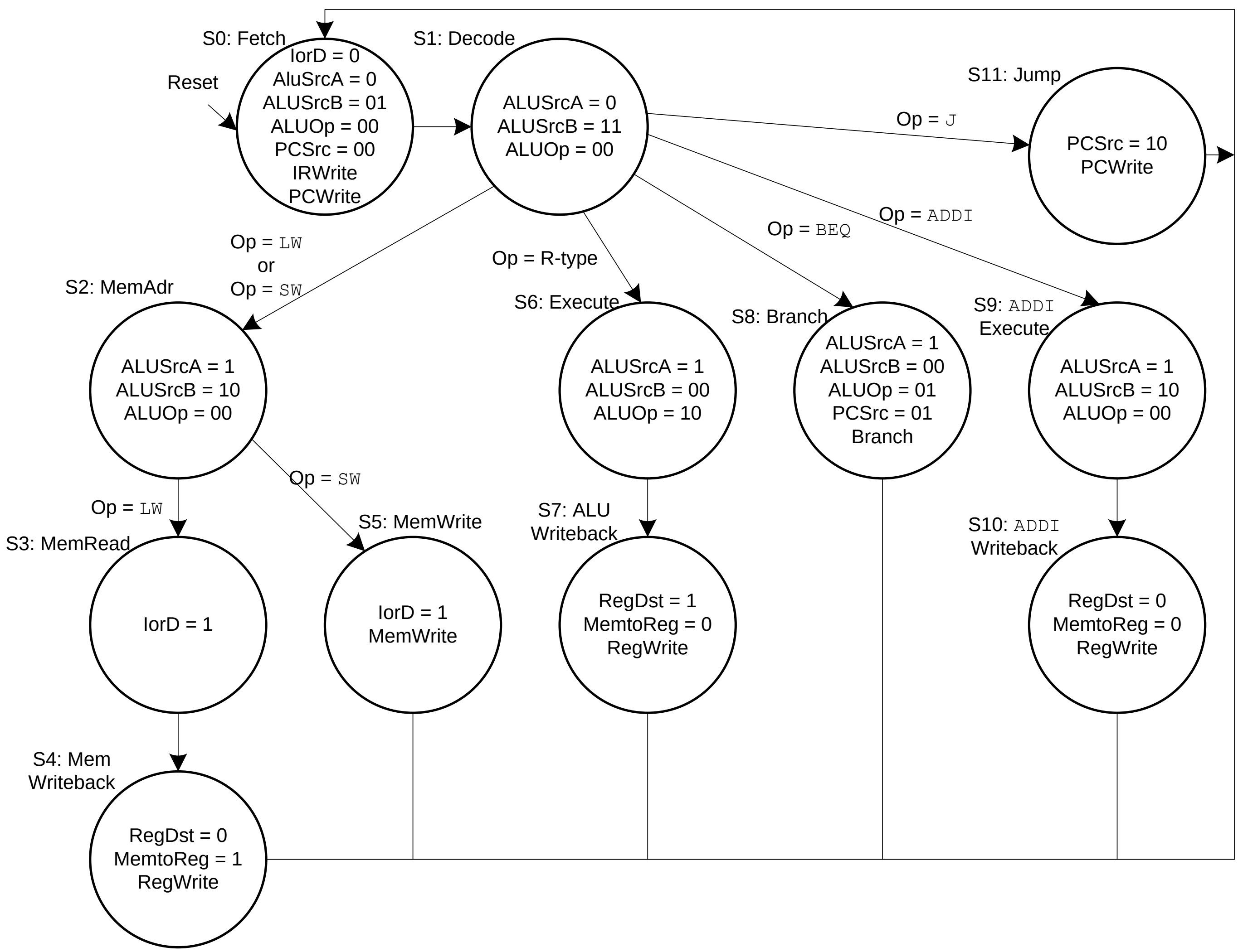

Figure 3: Multicycle MIPS controller FSM

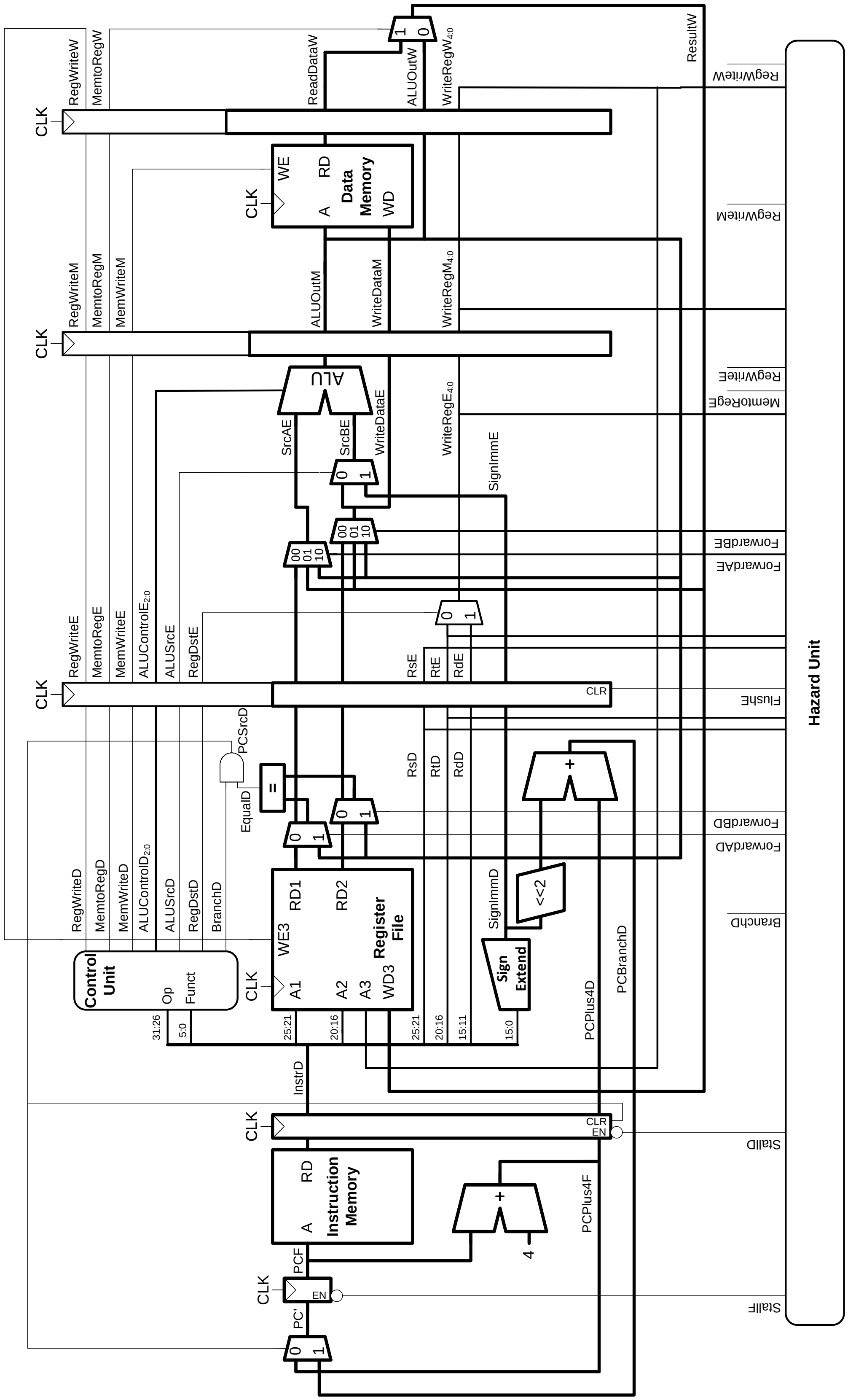

Figure 4: Pipelined MIPS Processor

MIPS Reference Cheat Sheet

INSTSTRUCTION SET (SUBSET)

Name (format, op, funct)	Syntax	Operation
add (R,0,32)	`add rd,rs,rt`	reg(rd) := reg(rs) + reg(rt);
add immediate (I,8,na)	`addi rt,rs,imm`	reg(rt) := reg(rs) + signext(imm);
add immediate unsigned (I,9,na)	`addiu rt,rs,imm`	reg(rt) := reg(rs) + signext(imm);
add unsigned (R,0,33)	`addu rd,rs,rt`	reg(rd) := reg(rs) + reg(rt);
and (R,0,36)	`and rd,rs,rt`	reg(rd) := reg(rs) & reg(rt);
and immediate (I,12,na)	`andi rt,rs,imm`	reg(rt) := reg(rs) & zeroext(imm);
branch on equal (I,4,na)	`beq rs,rt,label`	if reg(rs) == reg(rt) then PC = BTA else NOP;
branch on not equal (I,5,na)	`bne rs,rt,label`	if reg(rs) != reg(rt) then PC = BTA else NOP;
jump and link register (R,0,9)	`jalr rs`	$ra := PC + 4; PC := reg(rs);
jump register (R,0,8)	`jr rs`	PC := reg(rs);
jump (J,2,na)	`j label`	PC := JTA;
jump and link (J,3,na)	`jal label`	$ra := PC + 4; PC := JTA;
load byte (I,32,na)	`lb rt,imm(rs)`	reg(rt) := signext(mem[reg(rs) + signext(imm)]$_{7:0}$);
load byte unsigned (I,36,na)	`lbu rt,imm(rs)`	reg(rt) := zeroext(mem[reg(rs) + signext(imm)]$_{7:0}$);
load upper immediate (I,14,na)	`lui rt,imm`	reg(rt) := concat(imm, 16 bits of 0);
load word (I,35,na)	`lw rt,imm(rs)`	reg(rt) := mem[reg(rs) + signext(imm)];
multiply, 32-bit result (R,28,2)	`mul rd,rs,rt`	reg(rd) := reg(rs) * reg(rt);
nor (R,0,39)	`nor rd,rs,rt`	reg(rd) := not(reg(rs) | reg(rt));
or (R,0,37)	`or rd,rs,rt`	reg(rd) := reg(rs) | reg(rt);
or immediate (I,13,na)	`ori rt,rs,imm`	reg(rt) := reg(rs) | zeroext(imm);
set less than (R,0,42)	`slt rd,rs,rt`	reg(rd) := if reg(rs) < reg(rt) then 1 else 0;
set less than unsigned (R,0,43)	`sltu rd,rs,rt`	reg(rd) := if reg(rs) < reg(rt) then 1 else 0;
set less than immediate (I,10,na)	`slti rt,rs,imm`	reg(rt) := if reg(rs) < signext(imm) then 1 else 0;
set less than immediate unsigned (I,11,na)	`sltiu rt,rs,imm`	reg(rt) := if reg(rs) < signext(imm) then 1 else 0;
shift left logical (R,0,0)	`sll rd,rt,shamt`	reg(rd) := reg(rt) << shamt;
shift left logical variable (R,0,4)	`sllv rd,rt,rs`	reg(rd) := reg(rt) << reg(rs$_{4:0}$);
shift right arithmetic (R,0,3)	`sra rd,rt,shamt`	reg(rd) := reg(rt) >>> shamt;
shift right logical (R,0,2)	`srl rd,rt,shamt`	reg(rd) := reg(rt) >> shamt;
shift right logical variable (R,0,6)	`srlv rd,rt,rs`	reg(rd) := reg(rt) >> reg(rs$_{4:0}$);
store byte (I,40,na)	`sb rt,imm(rs)`	mem[reg(rs) + signext(imm)]$_{7:0}$:= reg(rt)$_{7:0;}$
store word (I,43,na)	`sw rt,imm(rs)`	mem[reg(rs) + signext(imm)] := reg(rt);
subtract (R,0,34)	`sub rd,rs,rt`	reg(rd) := reg(rs) - reg(rt);
subtract unsigned (R,0,35)	`subu rd,rs,rt`	reg(rd) := reg(rs) - reg(rt);
xor (R,0,38)	`xor rd,rs,rt`	reg(rd) := reg(rs) ^ reg(rt);
xor immediate (I,14,na)	`xori rt,rs,imm`	reg(rt) := rerg(rs) ^ zeroext(imm);

Definitions

- Jump to target address: JTA = concat((PC + 4)$_{31:28}$, address(label), 00$_2$)
- Branch target address: BTA = PC + 4 + imm * 4

Clarifications

- All numbers are given in decimal form (base 10).
- Function signext(x) returns a 32-bit sign extended value of x in two's complement form.
- Function zeroext(x) returns a 32-bit value, where zero are added to the most significant side of x.
- Function concat(x, y, …, z) concatenates the bits of expressions x, y, …, z.
- Subscripts, for instance X$_{8:2}$, means that bits with index 8 to 2 are spliced out of the integer X.
- Function *address*(x) means the address of label x.
- NOP and na means "no operation" and "not applicable", respectively.
- shamt is an abbreviation for "shift amount", i.e. how much bit shifting that should be done.

INSTRUCTION FORMAT

R-Type

31–26	25–21	20–16	15–11	10–6	5–0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	

I-Type

31–26	25–21	20–16	15–0
op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

J-Type

31–26	25–0
op	address
6 bits	26 bits

REGISTERS

Name	Number	Description
$zero	0	constant value 0
$at	1	assembler temp
$v0	2	function return
$v1	3	function return
$a0	4	argument
$a1	5	argument
$a2	6	argument
$a3	7	argument
$t0	8	temporary value
$t1	9	temporary value
$t2	10	temporary value
$t3	11	temporary value
$t4	12	temporary value
$t5	13	temporary value
$t6	14	temporary value
$t7	15	temporary value
$s0	16	saved temporary
$s1	17	saved temporary
$s2	18	saved temporary
$s3	19	saved temporary
$s4	20	saved temporary
$s5	21	saved temporary
$s6	22	saved temporary
$s7	23	saved temporary
$t8	24	temporary value
$t9	25	temporary value
$k0	26	reserved for OS
$k1	27	reserved for OS
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

MIPS Reference Cheat Sheet

By David Broman
KTH Royal Institute of Technology

If you find any errors or have any feedback on this document, please send me an email: dbro@kth.se

Version 1.0, December 19, 2014