Final Project Proposal

Multicycle design for MIPS like Machine

Vinod Halaharvi

For the final project I am proposing a computer with the processor type and Instruction set architecture that is subset of MIPS instruction set. For this proposal I will be using Hardware Multiplier as the novel feature.

In design document we will see step by step of how to implement a simple computer that uses a very minimal MIPS instruction set architecture to perform computation on integer data types. Floating-point operations are not supported as a part of this project. This machine will have a 32-bit word size. The machine will implement R-class, load/store, beq, and J type instructions. Just like MIPS I our machine will have three different instruction formats R, I and J format

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | Format | 31 – 26 | 25 – 21 | 20 – 15 | 15 – 11 | 10 – 6 | 5 – 0 | | R | 6 BIT OPC | rs | rt | rd | shamt | funct | | I | 6 BIT OPC | rs | rt | Immediate | | | | J | 6 BIT OPC | address | | | | | |

The flow of information in this design document is as follows - We first discuss on a very high level, the ISA for the machine. We then show a basic (not yet complete) flow chart to implement this micro-architecture. We then move on to implement the design. During the implementation we have two major logical separation of concerns, viz. data path design and control path design. The design schematic is very similar to what’s discussed in the multicycle design of the class. So we skip over the data path design and focus more on the control path that controls how data flows from one cycle to another. We then discuss control path and how design a sequencer that controls it. We design sequencer using finite state machine (FSM). Our machine uses a multicycle data path.

Before we dive into any further, the basic information about our machine is as follows. ISA and Micro-architecture is same (but a minimal subset of) as MIPS machine. The basic word size of the machine is 32 bits and all registers are 32 bit general-purpose registers. There are no special general-purpose registers (register zero could be thought of special), though we apply some convention that discourages the use of some registers. There will be 5 bits dedicated to address the register. So we have 32 different registers for this machine. Apart from these general-purpose registers, some special registers are only accessed by the machine and are not accessible by the programmer. Those registers are Program Counter (PC), IR (Instruction register), all the multiplexers, ALU, ALU controller and the main controller also called the sequencer. Data path width is shown in the architecture diagram where relevant. Memory access is allowed only by store and load instructions. The PC is not allowed to be access by the user as one of the general-purpose register. This machine does not have any condition codes. All I/O is performed using memory mapped I/O. All instructions are of fixed size. Though there will be an over flow protection for our machine, it not completely defined yet. Depending on the operation codes we can divide the instructions to the following classes.

|  |  |
| --- | --- |
| Instruction Class | OpCode |
| r-class | 000000 |
| lw | 100011 |
| sw | 101011 |
| beq | 000100 |
| j | 000010 |

## Novel Feature

The novel feature to this project will be Hardware Multiplier. To fast multiply two numbers we will use Algorithm from Patterson and Hennessey, page 233.

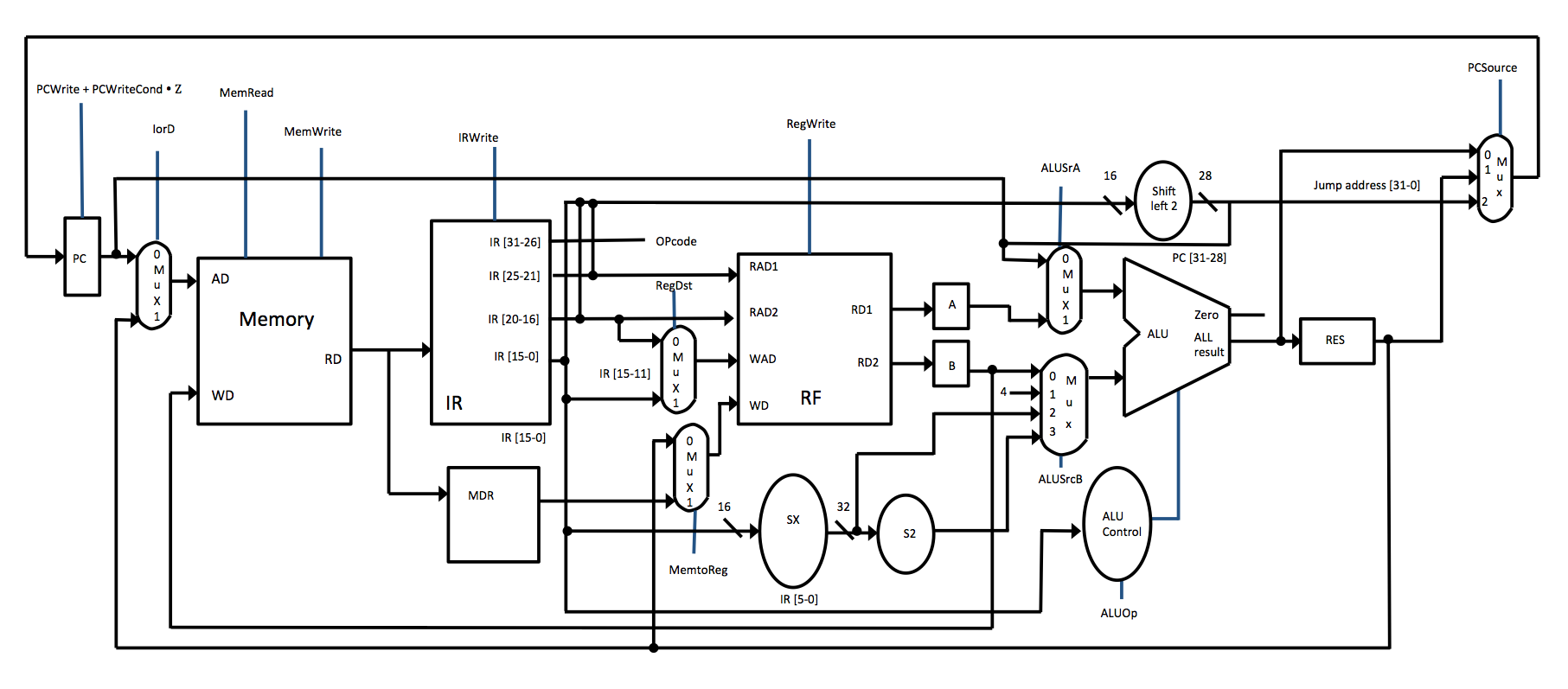
## Multicycle instruction design

Even though single cycle design is simpler to implement, it is not very efficient. Since each instruction cycle takes only one clock cycle, the clock period has to be long enough to accommodate the slowest instruction. Usually the slowest instructions are Memory fetch/store, divide and multiply. In a single cycle design there is one state component, which is updated at the rising edge of every clock cycle and the rest is combinational. Since there is only one control state, the design of the sequencer is fairly simple for single cycle design. We can convert a single cycle design to a multicycle design by introducing state components (Flip Flops and registers, etc.) with in the combinational circuitry. Though this increase the CPI, the clock period is much smaller, since we reduced the critical path delay in our combinational circuitry. So to convert a single cycle design to a multicycle design, the data path remains mostly the same. But there is a new overhead of coming up with a controller/sequencer that keeps track of the state transitions. We use an FSM for our machine to sequence the control actions.

## Block diagram

REFERENCE: At this time, we only discuss as very basic control design. But as the class progresses, we will improve the controller to add more instructions. The control design below was inspired by set of YouTube videos from Prof. Anshul Kumar. Link: <https://www.youtube.com/watch?v=Ngu1UbRAeqQ>

REFERENCE: Class notes, http://sites.fas.harvard.edu/~cscie287/fall2014/MIPS%20Multicycle%20Data%20Path.png



All blue lines are the outputs from the controller. Controller takes only one input, opcode of the instruction, which is IR[5-0]. For ALU Control, the input is ALUOp from the controller and the opcode. The output is a four-bit operation to be performed by ALU as shown below

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction opcode** | **ALUOp** | **Instruction operation** | **Funct Field** | **Desired ALU action** | **ALU control input** |
| LW | 00 | load word | xxxxxx | add | 0010 |
| SW | 00 | store word | xxxxxx | add | 0010 |
| BEQ | 01 | branch equal | xxxxxx | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| R-type | 10 | subtract | 100010 | subtract | 0110 |
| R-type | 10 | AND | 100100 | and | 0000 |
| R-type | 10 | OR | 100101 | or | 0001 |
| R-type | 10 | set on less than | 101010 | set on less than | 0111 |

## control path design

REFERENCE: At this time, we only discuss as very basic control design. But as the class progresses, we will improve the controller to add more instructions. The control design below was inspired by set of YouTube videos from Prof. Anshul Kumar. Link: <https://www.youtube.com/watch?v=Ngu1UbRAeqQ>

For a multicycle design we need to identify what steps needs to be performed in which cycle. In Multicycle design, each instruction could take multiple cycles. Each part of instruction that runs in one cycle needs to be identified. We call this micro operation for that instruction. For example for an r-class instruction, the micro operations are Instruction Fetch, PC Increment, Instruction Decode, Register read, ALU Operation and Register Write. Splitting each instruction this way in to micro operations help us refactoring micro operations of the machine and reduces the complexity of our implementation. Further more, the micro operations are not specific to each instruction as many instructions share common set of micro operations.

For example for add and subtract the micro operations are the same. What differentiate them are the control signals. For both these instructions we have to do an Instruction fetch, Instruction Decode, Register Read, ALU Operation and Register Write. The following below shows micro operations for each instruction class for our machine. The micro operations that can happen in the same clock cycle are shown together in a single box. For example below, Instruction read and PC increment can both happen in single clock cycle. Similarly both registers A and B can be incremented in a single clock cycle.

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

Firstly we define control signals for each micro operation. Then we come up with state transition diagram for our controller. The input to the state transition diagram is the opcode of the instruction and the current control state. Based on this, the new control state is determined. Once the machine is booted, it cycles through control states over and over defined by the rules of our FSM until it is turned off. Shown below, are the control signals for each micro operation. We group micro operations in to logical subsets as shown below.

PC Group

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Micro Operation | PCWriteCond | PCWrite | PCSource | Shortname |
| PC = PC + 4 | 1 | X | 1 | PCInc |
| If (A == B) PC = Res | 0 | 1 | 0 | branch |
| PC = PC[31-28] || s2(IR[25-0]) | 1 | X | 2 | jump |
| Default | 0 | 0 | X | nop |

RF Group

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Micro operation | RegWrite | RegDst | MemtoReg | Shortname |
| A = RF[IR[25-21]] | 0 | X | X | rs2A |
| B = RF[IR[20-16] | 0 | X | X | rt2B |
| RF[IR[15-11]] = Res | 1 | 1 | 0 | res2rd |
| RF[IR[20-16]] = DR | 1 | 0 | 1 | mem2rt |
| RF[31] = res | 1 | 0 | 0 | res2\_31 |
| Default | 0 | X | X | nop |

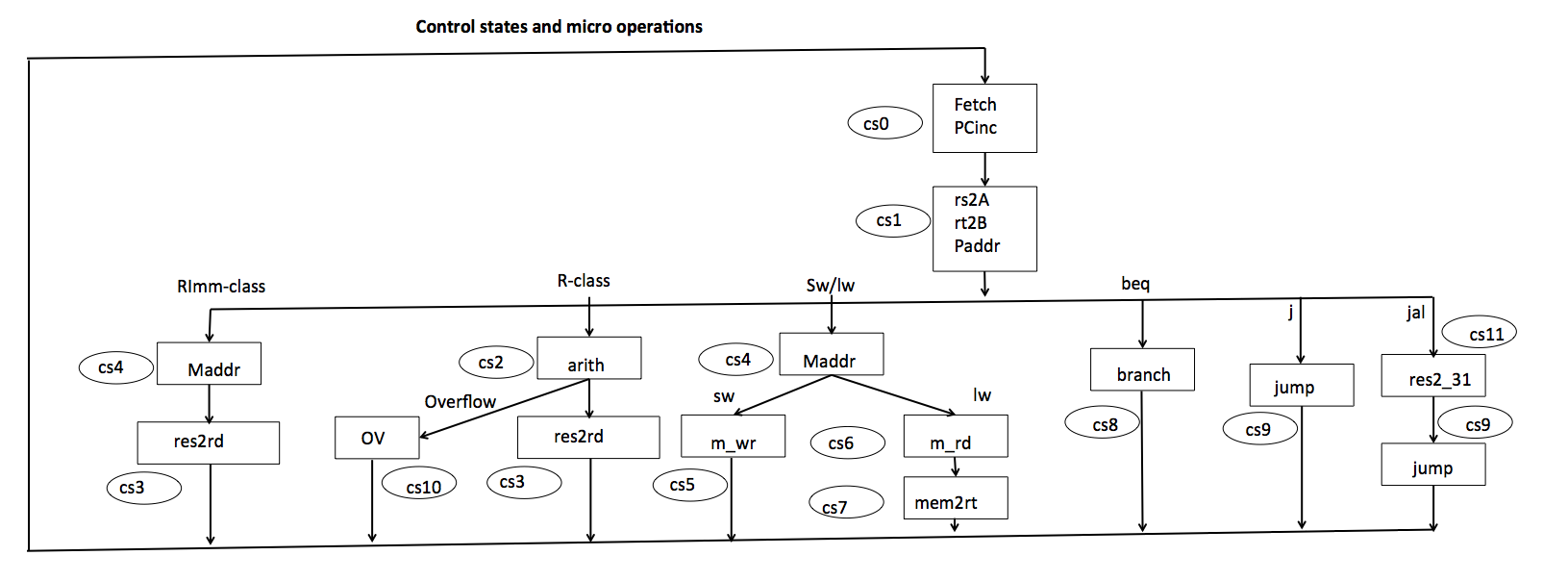
Memory Group

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Micro operation | MemWrite | MemRead | IorD | IRWrite | MDWrite | Shortname |
| IR = Mem[PC] | 0 | 1 | 0 | 1 | 0 | fetch |
| DR = Mem[Res] | 0 | 1 | 1 | 0 | 1 | m\_rd |
| Mem[Res] = B | 1 | 0 | 1 | 0 | 0 | m\_wr |
| Default | 0 | 0 | X | 0 | 0 | nop |

ALU Group

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Micro operation | ALUOp | ALUSrcA | ALUSrcB | RegWrite | Shortname |
| PC = PC + 4 | 0 | 0 | 1 | 0 | PCInc |
| Res = A op B | 2 | 1 | 0 | 1 | arith |
| Res = A + sx(IR[15-0]) | 0 | 1 | 2 | 1 | Maddr |
| Res = PC + s2(sx(IR[15-11])) | 0 | 0 | 3 | 1 | Paddr |
| If (A ==B) PC = Res | 1 | 1 | 0 | 0 | branch |
| Default | X | X | X | 0 | nop |

Refactoring the microinstructions we arrive at the figure as shown below.

****

By looking at the above figure we can fill in the details of Control State transition matrix as shown below. They way to read this is, if we are in state cs0, then regardless of what the opcode of the current instruction is, we always transition to state cs1. And if we are in state cs1 we transition to any of these states - cs4, cs2, cs8, cs9 or cs11 depending on opcode. So for example, if we are in cs1 and the opcode of the current instruction is sw then we transition to state cs4.

|  |  |  |
| --- | --- | --- |
| current control state | class instruction | next control state |
| cs0 | x | cs1 |
| cs1 | r-class | cs2 |
| cs1 | sw/lw | cs4 |
| cs1 | beq | cs8 |
| cs1 | j | cs9 |
| cs1 | jal | cs11 |
| cs1 | r imm | cs4 |
| cs2 | x | cs3 |
| cs2 | overflow | cs10 |
| cs3 | x | cs0 |
| cs4 | sw | cs5 |
| cs4 | lw | cs6 |
| cs5 | x | cs0 |
| cs6 | x | cs7 |
| cs7 | x | cs0 |
| cs8 | x | cs0 |
| cs9 | x | cs0 |
| cs10 | x | cs0 |
| cs9 | x | cs0 |

By using the opcode for each instruction class we can re-write the table as shown below.

|  |  |  |
| --- | --- | --- |
| Current Control state | Class instruction | Next Control State |
| 0000 | XXXXXX | 0001 |
| 0001 | 000000 | 0010 |
| 0001 | 10X011 | 0100 |
| 0001 | 000100 | 1000 |
| 0001 | 000010 | 1001 |
| 0001 | XXXXXX | 1011 |
| 0001 | any Imm instr | 0100 |
| 0010 | XXXXXX | 0011 |
| 0010 | exception | 1010 |
| 0011 | 100011 | 0000 |
| 0100 | XXXXXX | 0101 |
| 0100 | XXXXXX | 0110 |
| 0101 | XXXXXX | 0000 |
| 0110 | XXXXXX | 0000 |

|  |  |  |
| --- | --- | --- |
| 0111 | XXXXXX | 0000 |
| 1000 | XXXXXX | 0000 |
| 1001 | XXXXXX | 0000 |
| 1010 | XXXXXX | 0000 |
| 1011 | XXXXXX | 0000 |

## Assembly language for our machine (Work in progress):

REFERENCE: http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm

### Registers

There are 32 general-purpose registers. Registers are prefixed with '$' at the front.

Registers can either be addressed using register names, or register numbers

Register-addressing using register numbers: $0 - $31

Register addressed using register names: $v0 , $a0 etc.

Programmer can access all the general-purpose registers any way he likes. But we recommend conventions as shown in Goodman and Miller book.

### Program

Program contains simple text and is stored in a text file. Program has a data section and a text section.

'.data' directive is used to declare and define data. '.text' directive is used for program code.

Variables are declared and allocated in the .data section.

.text section contains executable instructions.

### Text Section

.text directive is used for program code

program execution starts at 'main:'

anything after # is considered a comment and will be ignored by the assembler

### Data Section

Variables are declared and assigned in the .data section.

### Variables

variable space is allocated in main memory and all variables have types. Variables are declared/defined as shown below.

name: type value(s)

### Examples

foo: .word 0 # integer variable initialized to 0

bar: .byte 'h', 'e', 'l', 'l', 'o' # character array with 'hello' character elements

array: .space 10 # allocate 10 consecutive bytes.

### Load/Store Instructions

load:

lw $t0, MemAddress # load a word in memory from MemAddress

store:

sw $t0, MemAddress # store a word to memory at MemAddress

### Example of simple program

.data

foo: .word 0

bar: .word 1

.text

main:

lw $t0, foo

lw $t1, bar

add $t2, $t0, $t1

done

### Control Structures

j target # Unconditional jump to effective address target

jal target # Unconditional jump to effective address after storing the PC+4 in to $31

Please refer to the ISA section in this document for other instructions

# Instruction Set

Instruction Set for our machine is as shown below:

REFERENCE: Mips IV instruction manual

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Mnemonic | 31 - 26 | 25 - 21 | 21 – 16 | 15 – 11 | 10 – 6 | 5 – 0 | Description |
| add | 000000 | rs | rt | rd | 00000 | 100000 | The 32-bit word value in GPR rt is added to the 32-bit value in GPR rs and the 32-bit arithmetic result is placed into GPR rd |
| addi | 011000 | rs | rt | imm | imm | imm | The 16-bit signed immediate is added to the 32-bit value in GPR rs and the 32-bit arithmetic result is placed into GPR rt. |
| and | 000000 | rs | rt | rd | 00000 | 100100 | The contents of GPR rs are combined with the contents of GPR rt in a bitwise logical AND operation. The result is placed into GPR rd |
| andi | 001100 | rs | rt | imm | imm | imm | The 16-bit immediate is zero-extended to the left and combined with the contents of GPR rs in a bitwise logical AND operation. The result is placed into GPR rt |
| beq | 000100 | rs | rt | offset | offset | offset | An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) to form a PC-relative effective target address. If the contents of GPR rs and GPR rt are equal, branch to the effective target address |
| bne | 000101 | rs | rt | offset | offset | offset | An 18-bit signed offset (the 16-bit offset field shifted left 2 bits) is added to the address of the instruction following the branch (not the branch itself) to form a PC-relative effective target address.If the contents of GPR rs and GPR rt are not equal, branch to the effective target address |
| j | 000010 | instr\_index | instr\_index | instr\_index | instr\_index | instr\_index | The low 28 bits of the target address is the instr\_index field shifted left 2 bits and is overlayed on PC, by retaining PC's most significant 4 bits to form effective target address |
| jal | 000011 | instr\_index | instr\_index | instr\_index | instr\_index | instr\_index | Place the return address link in GPR 31.The low 28 bits of the target address is the instr\_index field shifted left 2 bits and is overlayed on PC, by retaining PC's most significant 4 bits to form effective target address. Jump to the effective target address. |
| lw | 100011 | base | rt | offset | offset | offset | The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR rt. The 16-bit signed offset is added to the contents of GPR base to form the effective address. |
| nor | 000000 | rs | rt | rd | 00000 | 100111 | The contents of GPR rs are combined with the contents of GPR rt in a bitwise logical NOR operation. The result is placed into GPR rd |
| xor | 000000 | rs | rt | rd | 00000 | 100110 | The contents of GPR rs are combined with the contents of GPR rt in a bitwise logical XOR operation. The result is placed into GPR rd. |
| or | 000000 | rs | rt | rd | 00000 | 100101 | The contents of GPR rs are combined with the contents of GPR rt in a bitwise logical OR operation. The result is placed into GPR rd. |
| ori | 001101 | rs | rt | imm | imm | imm | The 16-bit immediate is zero-extended to the left and combined with the contents of GPR rs in a bitwise logical OR operation. The result is placed into GPR rt. |
| slt | 000000 | rs | rt | rd | 00000 | 101010 | Compare the contents of GPR rs and GPR rt as signed integers and record the Boolean result of the comparison in GPR rd. If GPR rs is less than GPR rt the result is 1 (true), otherwise 0 (false). |
| sll | 000000 | rs | rt | rd | sa | 000000 | The contents of the 32-bit word of GPR rt are shifted left, inserting zeroes into the emptied bits; the word result is placed in GPR rd. The bit shift count is specified by sa. |
| srl | 000000 | rs | rt | rd | sa | 000010 | The contents of the 32-bit word of GPR rt are shifted right, inserting zeroes into the emptied bits; the word result is placed in GPR rd. The bit shift count is specified by sa |
| sra | 000000 | rs | rt | rd | sa | 000011 | The contents of the 32-bit word of GPR rt are shifted right, duplicating the sign-bit (bit 31) in the emptied bits; the word result is placed in GPR rd. The bit shift count is specified by sa |
| sub | 000000 | rs | rt | rd | 00000 | 100010 | The 32-bit word value in GPR rt is subtracted from the 32-bit value in GPR rs and the 32-bit arithmetic result is placed into GPR rd. |
| sw | 101011 | base | rt | offset | offset | offset | The least-significant 32-bit word of register rt is stored in memory at the location specified by the aligned effective address. The 16-bit signed offset is added to the contents of GPR base to form the effective address. |

## APPENDIX A

|  |  |
| --- | --- |
| Abbreviations | |
| ISA | Instruction set architecture |
| FSM | Finite State Machine |
| PC | Program counter |
| IR | Instruction Register |
| MemDR | Memory Data Register |
| RF | Register File |