Single-Cycle Processor Datapath & Control

Adding Instructions (esp. Jumps and Branches) to the MIPS ISA

Project Assignment & Description

Your project team will extend the MIPS ISA by implementing a number of additional instructions. These instructions, plus some existing MIPS instructions (which are part of the ISA) will be added to the single-cycle processor implementation. To do this requires modifying both the datapath and the control. You are to modify and extend these as necessary, so that the following 35 instructions are implemented:

R-format: add, and, balrn, balrz, brn, brz, jalr, jr, nor, or, slt, sub

I-format: addi, andi, balmn, balmz, beq, beqal, bmn, bmz, bne, bneal, jalm, jalpc, jm, jpc, lw, ori, sw

J-format: baln, balz, bn, bz, jal, j

Your Task and Assumptions

You must determine which of the above instructions are actual MIPS instructions (in the current ISA), and which ones are new additions. You also must determine which of the actual MIPS instructions are already implemented in the 'MIPS-lite' processor (Verilog model- in attachment) that has been given. Then you must design the revised single-cycle datapath and revised control units which make a processor that executes all 35 instructions.

After designing the new enhanced processor, **you will implement it in Verilog HDL.**

You may assume that the real MIPS instructions are correctly specified in attachment word file.

Additional Jumps and Branches

MIPS is a particularly minimalist RISC ISA. It seeks to implement in hardware only instructions that are absolutely necessary, which means that many operations are possible only by using two or more MIPS instructions. Other RISC architectures take a looser approach to additional instructions. Many have a richer array of choices for jumping and branching (see PowerPC for example).

Control instructions (jumping/branching) have 3 aspects: whether the change of control is conditional or unconditional, whether a return address link is stored or not, and which option for the target address is chosen.

Based on the above, there are several more useful instructions for control change that could be added to MIPS. The target address could be obtained indirectly, by loading it from memory. The branches could have a return link, or not have one, just as the jumps do. The conditions that branching is based on do not have to be limited to the values in two registers (or a register and $zero) specified in the branch instruction. Many architectures use a Status register to save the ALU status bits from the previous instruction (even including LT, GT, LE, GE, EQ, and NE after compare operations, as well as Zero, Negative, oVerflow, and Carry bits resulting from arithmetic operations)

In the MIPS we have studied so far, branches change control conditionally, and jumps change it unconditionally. MIPS branches never store a return link, but jumps may link (e.g. jal) or may not (e.g. j). MIPS calculates the target address in 3 ways: PC-relative (based on 16-bit immediate field, as in beq/bne), pseudo-direct (based on 26-bit immediate field, as in j/jal) and register direct (the target address is stored in a register, as in jr/jalr

A major focus of this project is to add a number of new jump and branch instructions, to enrich the MIPS instruction set. The following is a list and description of the 18 new instructions which are not part of official MIPS, but which you will add in this project.

Specifications of New Instructions

Unconditional change of control (jumps)

Instr	Type	Code	Syntax	Meaning
jm	I-type	opcode=18	jm imm16($rs)	jumps to address found in memory (indirect jump)
jalm	I-type	opcode=19	jalm $rt, imm16($rs)	jumps to address found in memory (indirect jump), link address is stored in $rt (which defaults to 31)
jpc	I-type	opcode=30	jpc Target	jumps to PC-relative address (formed as beq and bne do)
jalpc	I-type	opcode=31	jalpc $rt, Target	jumps to PC-relative address (formed as beq and bne do), link address is stored in $rt (which defaults to 31)

Conditional change of control (branches)

Instr	Type	Code	Syntax	Meaning
bmz	I-type	opcode=20	bmz imm16($rs)	if Status [Z] = 1, branches to address found in memory
bmn	I-type	opcode=21	bmn imm16($rs)	if Status [Z] = 0, branches to address found in memory
balmz	I-type	opcode=22	balmz $rt, imm16($rs)	if Status [Z] = 1, branches to address found in memory, link address is stored in $rt (which defaults to 31)
balmn	I-type	opcode=23	balmn $rt, imm16($rs)	if Status [Z] = 0, branches to address found in memory, link address is stored in $rt (which defaults to 31)
brz	R-type	funct=20	brz $rs	if Status [Z] = 1, branches to address found in register $rs
brn	R-type	funct=21	brn $rs	if Status [Z] = 0, branches to address found in register $rs

balrz	R-type	funct=22	balrz $rs, $rd	if Status [Z] = 1, branches to address found in register $rs link address is stored in $rd (which defaults to 31)
balrn	R-type	funct=23	balrn $rs, $rd	if Status [Z] = 0, branches to address found in register $rs link address is stored in $rd (which defaults to 31)
bz	J-type	opcode=24	bz Target	if Status [Z] = 1, branches to pseudo-direct address (formed as j does)
bn	J-type	opcode=25	bn Target	if Status [Z] = 0, branches to pseudo-direct address (formed as j does)
balz	J-type	opcode=26	balz Target	if Status [Z] = 1, branches to pseudo-direct address (formed as jal does), link address is stored in register 31
baln	J-type	opcode=27	baln Target	if Status [Z] = 0, branches to pseudo-direct address (formed as jal does), link address is stored in register 31
beqal	I-type	opcode=44	beqal $rs, $rt, Target	if R[rs] = R[rt], branches to PC-relative address (formed as beq & bne do), link address is stored in register 31
bneal	I-type	opcode=45	bneal $rs, $rt, Target	if R[rs] != R[rt], branches to PC-relative address (formed as beq & bne do), link address is stored in register 31

<u>Status Register</u>

Some of the conditional branches test the Z bit in the Status register. So the MIPS datapath will need to have a Status register, with the following 4 bits: Z (if the ALU result is zero), N (is the ALU result is negative), V (if the ALU result overflowed) and C (if the ALU result caused a carry out from the MSB). The Status register will be loaded with the ALU results each clock cycle.

Only the Z bit is used in the new conditional branches, but the rest are available for future expansion of the ISA.

The real MIPS machine language

As noted earlier, many of the instructions in the MIPS assembly language are actually pseudoinstructions. They do not correspond directly to machine instructions, but instead are translated by the assembler into true machine instructions with different opcodes, or into sequences of two or three instructions. We will look at several examples.

1. li (load immediate)

```
    li    rd, imm
```

is translated into

```
    addiu rd, $0, imm
```

2. la (load address)

```
    la    rd, symbol
```

is translated into

```
    lui   $1, addr31-16      #  addr is the address of symbol
    ori   rd, $1, addr15-0


    la   rd, symbol(indexreg)
```

is translated into

```
    lui   $1, addr31-16      #  addr is the address of symbol
    ori   rd, $1, addr15-0
    add   rd, $1, indexreg
```

3. lw (load word)

At the machine level, the memory operand of the lw instruction is specified with a base register and a 16-bit offset.

```
    lw   rd, symbol
```

is translated into

```
    lui  $1, addr31-16      #  addr is the address of symbol
    lw   rd, addr15-0($1)
    lw   rd, symbol(indexreg)
```

is translated into

```
    lui  $1, addr31-16      #  addr is the address of symbol
    addu $1, $1, indexreg
    lw   rd, addr15-0($1)
```

4. multiplication and division

The difficulty in implementing multiplication is that the product of two 32-bit integers may be 64 bits long.

This is handled at the machine level by placing the result of every multiply operation in two special 32-bit registers called HI and LO (for the high-order and low-order bits of the result, respectively). Two machine instructions (mfhi and mflo) are provided to copy the contents of HI and LO to general registers. The true multiply instruction has two register operands:

```
    mult   rs, rt   #  HI, LO <-- rs * rt

    mfhi   rd       #   rd <-- HI
    mflo   rd       #   rd <-- LO
```

Using HI and LO, the pseudoinstruction

```
    mul   rd, rs, rt
```

is translated into

```
    mult   rs, rt
    mflo   rd
```

Note that only the low-order 32 bits of the result are transferred to the destination register. If the result carries over into HI, it is the programmer's responsibility to decide what to do with it.

The difficulty in implementing integer division is that division produces two results, a quotient and a remainder.

On MIPS, this is handled by placing the quotient in LO and the remainder in HI.

So the machine-level divide instruction is defined as follows:

```
    div   rs, rt   #   LO <-- rs / rt, HI <-- rs % rt
```

The assembly language provides pseudoinstructions div and rem which follow the same format as the other arithmetic and logical instructions, with three register operands:

```
    div   rd, rs, rt
```

is translated into

```
    dir   rs, rt
    mflo   rd

    rem   rd, rs, rt
```

is translated into

```
    div   rs, rt
    mfhi   rd
```

5. branches

Not all the branch instructions we have been using are actually implemented at the machine level. The machine level branches are:

```
    bltz   rs, label
    bgez   rs, label
    blez   rs, label
    bgtz   rs, label
    beq   rs, rt, label
    bne   rs, rt, label
```

How are the other branches implemented?

```
      b     label

==>   bgez  $0, label

      bgt   rs, rt, label

==>   slt   $1, rt, rs
      bne   $1, $0, label

      ble   rs, rt, label
```

```
==>   slt   $1, rt, rs
      beq   $1, $0, label

      blt   rs, rt, label

==>   slt   $1, rs, rt
      bne   $1, $0, label

      bge   rs, rt, label

==>   slt   $1, rs, rt
      beq   $1, $0, label
```

Another problem with branches:

The destination address is a 32-bit address, but branches only contain a 16-bit immediate operand. (The rest of the bits are used for the opcode and register specifiers.)

In fact, all branches are "PC-relative". The instruction contains an *offset from the current program counter*. (Actually, it contains the offset divided by 4, since all instruction addresses are on a word boundary.)

For example, the instruction **beq rs, rt, offset** is implemented with the semantics

if rs == rt then pc <-- pc + (offset<<2)

The offset is interpreted as a 2's complement integer.

```
0x40006C   top:   beqz   $s0, exit


0x40009C   1020 FFF4      beq   $1, $0, top
```

Q: How to jump further away?

A: The instructions j and jal use a 26-bit address.

If this isn't good enough, use jr or jalr, which permit a 32-bit destination address stored in a register.

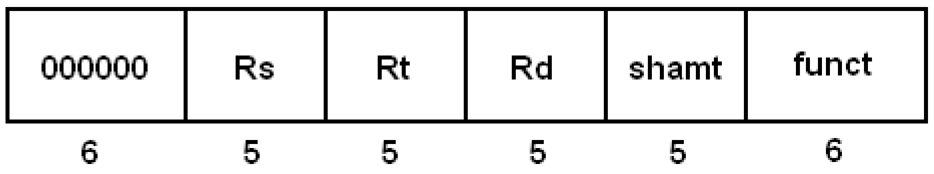
6. delayed branches and loads

Instruction formats

How are MIPS machine instructions encoded?

Each instruction is encoded in a 32-bit word, in one of 3 formats.

1. R format

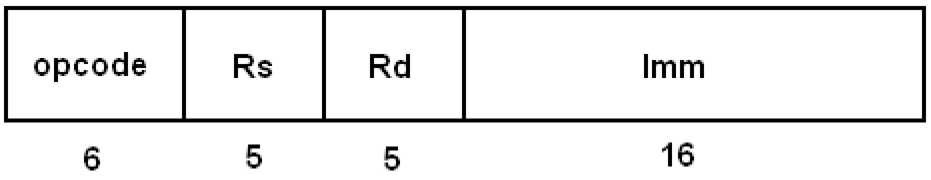


- shamt is "shift amount" (i.e., number of bits to shift), used in the immediate versions of the sll, srl, and sra instructions. In all other instructions in this format, the shamt field is 0.
- funct is "function", essentially an extension of the opcode which indicates which function is to be executed by this instruction.

used by:

- add, sub
- and, or, xor
- slt
- sll, srl
- mult, div (with Rd=0)
- jr (with Rt=Rd=0)
- jalr (with Rt=0)
- mflo, mfhi (with Rs=Rt=0)

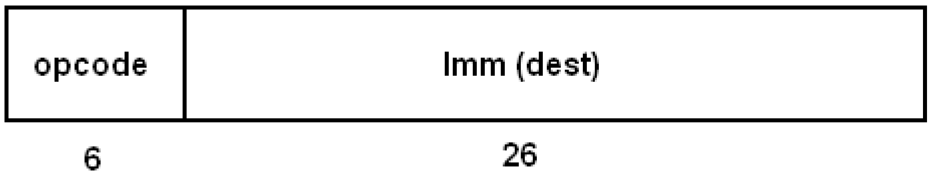
2. I format



used by:

- lw, sw, lb, sb
- addi, andi, ori, lui
- beq, bne, bltz, bgez, bgtz, blez

3. J format



used by: 100101

- j, jal

Mips opcodes

R format instructions (opcode 000000)

instruction	function code
add	100000
and	100100
div	011010
mult	011000
nor	100111
or	100101
sll	000000
sllv	000100
sra	000011
srav	000111
srl	000010
srlv	000110
sub	100010
xor	100110
slt	101010
jr	001000

jalr	001001

I format instructions

instruction	opcode
addi	001000
andi	001100
ori	001101
xori	001110
slti	001010
beq	000100
bgtz	000111
blez	000110
bne	000101
lw	100011
lb	100000
sw	101011
sb	101000

J format instructions

instruction	opcode
j	000010
jal	000011