RTL for a subset of the MIPS ISA using the Simulated Multicycle Implementation

In this simulation, most MIPS instructions are executed in a total of 4 clock cycles.

The first clock cycle is the same for all instructions, because it is during this cycle that the instruction is actually fetched from memory. (Note that, in this and subsequent examples, we may be able to do two micro-operations on the same clock.)

```
Cycle == 0:     IR ← M[PC], PC ← PC + 4
```

For the second and subsequent clock cycles, the micro-operations performed depend on the opcode of the instruction that is in the IR. In addition, in the RTL below `rs`, `rt`, `rd`, `func`, and `constant` are fields in the instruction which is in the IR.

Most branching instructions (`jr`, `beq`, `bne`, `j`) do nothing on cycles 2 and 3.

R-Type (all R-Format instructions other than `jr`)

```
Cycle == 1:     ALUInputA ← register[rs], ALUInputB ← register[rt]
Cycle == 2:     ALUOutput ← ALUInputA func ALUInputB (1)
Cycle == 3:     register[rd] ← ALUOuput

Note: (1) func is the function specified by the func field of the IR
```

`jr`

```
Cycle == 1:     PC ← register[rs]
```

`addi, andi, ori, xori, slti, lui`

```
Cycle == 1:     ALUInputA ← register[rs], ALUInputB ←  I constant (1)
Cycle == 2:     ALUOutput ← ALUInputA op ALUInputB                (2)
Cycle == 3:     register[rt] ← ALUOutput

Notes: (1) sign extended for addi,xori,slti; not sign-extended for andi,ori,lui
       (2) "op" is the appropriate operation based on the opcode
```

`lw, sw`

```
Cycle == 1:     ALUInputA ← register[rs], ALUInputB ← sign-extend(I constant)
Cycle == 2:     ALUOutput ← ALUInputA + ALUInputB
Cycle == 3 && opcode == lw: register[rt] ← M[ALUOutput]
Cycle == 3 && opcode == sw: M[ALUOutput] ← register[rt]
```

`beq, bne`

```
Cycle == 1:     (opcode == beq && register[rs] == register[rt] ||
                opcode == bne && register[rs] != register[rt]) :
                PC ← PC + sign-extend(I constant) * 4
```

`j`

```
Cycle == 1:     PC ← PC[31..28] | J constant * 4
```

`jal`

```
Cycle == 1:     ALUInputA ← PC, PC ← PC[31..28] | J constant * 4
Cycle == 2:     ALUOutput ← ALUInputA
Cycle == 3:     register[31] ← ALUOutput
```