RTL for a subset of the MIPS ISA using the Simulated Single Cycle Implementation

The micro-operations performed depend on the opcode of the instruction that is in the IR. (In the RTL below `rs`, `rt`, `rd`, `func`, and `constant` are fields in the instruction which is in the IR.) Each instruction ends with a clock cycle that performs the register loads and/or memory operation specified, including reading the next instruction into the IR. (Since the PC is loaded at the same time as the IR, the PC value that is used is the old value that was present at the start of the instruction)

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

```
PC ← PC + 4,
ALUOutput ← register[rs] func register[rt], (func is operation specified by func field of IR)
register[rd] ← ALUOutput,
IR ← M[PC] - end of this instruction and start of next
```

addi, andi, ori, xori, slti, lui

```
PC ← PC + 4,
register[rt] ← register[rs] op I constant, (constant is sign extended for addi,xori,slti;
    not sign-extended) for andi,ori,lui.  "op" is the operation specified by the op-code.)
IR ← M[PC] - end of this instruction and start of next
```

lw

```
PC ← PC + 4,
register[rt] ← M[register[rs] + sign-extend(I constant)],
IR ← M[PC] - end of this instruction and start of next
```

sw

```
PC ← PC + 4,
M[register[rs] + sign-extend(I constant)] ← register[rt],
IR ← M[PC] - end of this instruction and start of next
```

beq

```
if (register[rs] == register[rt]) PC + sign-extend(I constant) * 4 else PC ← PC + 4,
IR ← M[PC] - end of this instruction and start of next
```

bne

```
if (register[rs] != register[rt]) PC + sign-extend(I constant) * 4 else PC ← PC + 4,
IR ← M[PC] - end of this instruction and start of next
```

j

```
PC ← PC[31..28] | J constant * 4,
IR ← M[PC] - end of this instruction and start of next
```

jr

```
PC ← register[rs],
IR ← M[PC] - end of this instruction and start of next
```

jal

```
PC ← PC[31..28] | J constant * 4,
register[31] ← PC,
IR ← M[PC] - end of this instruction and start of next
```

jalr

```
PC <- register[rs],
register[31] ← PC,
IR ← M[PC] - end of this instruction and start of next
```

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 the RTL below, `rs`, `rt`, `rd`, `func`, and `constant` are fields in the instruction which is in the IR.

<u>R-Type (all R-Format instructions other than `jr`, `jalr`)</u>

```
Cycle == 1:     ALUInputA ← register[rs], ALUInputB ← register[rt]
Cycle == 2:     ALUOutput ← ALUInputA func ALUInputB (1)
Cycle == 3:     register[rd] ← ALUOutput
Note: (1) func is the function specified by the func field of the IR
```

<u>addi, andi, ori, xori, slti, lui</u>

```
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
```

<u>lw</u>

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

<u>sw</u>

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

<u>beq</u>

```
Cycle == 1:     (register[rs] == register[rt]) : PC ← PC + sign-extend(I constant) * 4
```

<u>bne</u>

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

<u>j</u>

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

<u>jr</u>

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

<u>jal</u>

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

jalr

```
Cycle == 1:     PC <- register[rs], ALUInputA <- PC
Cycle == 2:     ALUOutput ← ALUInputA
Cycle == 3:     register[31] ← ALUOutput
```