# MIPS instructions

<table>
<thead>
<tr>
<th>ALU</th>
<th>alu $rd,$rs,$rt</th>
<th>$rd = $rs &lt;alu&gt; $rt</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUi</td>
<td>alui $rd,$rs,value</td>
<td>$rd = $rs &lt;alu&gt; value</td>
</tr>
<tr>
<td>Data Transfer</td>
<td>lw $rt,offset($rs)</td>
<td>$rt = Mem[$rs + offset]</td>
</tr>
<tr>
<td></td>
<td>sw $rt,offset($rs)</td>
<td>Mem[$rs + offset] = $rt</td>
</tr>
<tr>
<td>Branch</td>
<td>beq $rs,$rt,offset</td>
<td>$pc = ($rd == $rs)? (pc+4+offset):(pc+4);</td>
</tr>
<tr>
<td>Jump</td>
<td>j address</td>
<td>pc = address</td>
</tr>
</tbody>
</table>
MIPS fixed sized instruction formats

R - Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
</table>

ALU

alu $rd,$rs,$rt

ALUi

alu $rt,$rs,value

Data Transfer

lw $rt,offset($rs)

sw $rt,offset($rs)

Branch

beq $rs,$rt,offset

Jump

j address

I - Format

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>value or offset</th>
</tr>
</thead>
</table>

J - Format

<table>
<thead>
<tr>
<th>op</th>
<th>absolute address</th>
</tr>
</thead>
</table>
Assembling Instructions

Suppose there are 32 registers, addu opcode=001001, addi op=001000

**alu $rd,$rs,$rt**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**addu $23, $0, $31**

0x00400020

```
001001:00000:11111:10111:00000:000000
```

**alui $rt,$rs,value**

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>value or offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00101:0000000000000101</td>
</tr>
</tbody>
</table>

**addi $17, $0, 5**

0x00400024

```
001000:00000:00101:0000000000000000101
```
MIPS instruction formats

Arithmetic

1. Immediate addressing
\[ \text{addi } $rt, $rs, \text{value} \]

2. Register addressing
\[ \text{add } $rd, $rs, $rt \]

3. Base addressing
\[ \text{lw} \ $rt, \text{offset($rs)} \]
\[ \text{sw} \ $rt, \text{offset($rs)} \]

Data Transfer

Conditional branch
\[ \text{beq } $rs, $rt, \text{offset} \]

Unconditional jump
\[ \text{j address} \]

Unconditional jump
\[ \text{j address} \]
# MIPS registers and conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Conventional usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>Expression evaluation &amp; function return</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Arguments 1 to 4</td>
</tr>
<tr>
<td>$t0-$t9</td>
<td>8-15,24,35</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>Saved Temporary (preserved across call)</td>
</tr>
<tr>
<td>$k0-$k1</td>
<td>26-27</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>Pointer to global area</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>Return address (used by function call)</td>
</tr>
</tbody>
</table>
C function to MIPS Assembly Language

```c
int power_2(int y) { /* compute x=2^y; */
    register int x, i; x=1; i=0; while(i<y) { x=x*2; i=i+1; }
    return x;
}
```

Assembly (.s)

```
addi $t0, $0, 1  # x=1;
addu $t1, $0, $0  # i=0;
w1:  bge $t1,$a0,w2  # while(i<y) { /* bge= greater or equal */
    addu $t0, $t0, $t0  # x = x * 2; /* same as x=x+x; */
    addi $t1,$t1,1  # i = i + 1;
    beq $0,$0,w1  # }
w2:  addu $v0,$0,$t0  # return x;
    jr $ra  # jump on register ( pc = ra; )
```

Comments

Exit condition of a while loop is if ( i >= y ) then goto w2
Power_2.s: MIPS storage assignment

Byte address, not word address

0x00400020  addi  $8, $0, 1  # addi $t0, $0, 1
0x00400024  addu  $9, $0, $0  # addu $t1, $0, $0
0x00400028  bge   $9, $4, 2  # bge $t1, $a0, w2
0x0040002c  addu  $8, $8, $8  # addi $t0, $t0, $t0
0x00400030  addi  $9, $9, 1  # addi $t1, $t1, 1
0x00400034  beq   $0, $0, -3  # beq $0, $0, w1
0x00400038  addu  $2, $0, $8  # addu $v0, $0, $t0
0x0040003c  jr     $31  # jr $ra

2 words after pc fetch
after bge fetch pc is 0x00400030
plus 2 words is 0x00400038

Byte address, not word address
### Machine Language Single Stepping

Assume `power2(0)` is called; then \( a0 = 0 \) and \( ra = 700018 \)

<table>
<thead>
<tr>
<th>$pc$</th>
<th>$v0$</th>
<th>$a0$</th>
<th>$t0$</th>
<th>$t1$</th>
<th>$ra$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00400020</td>
<td>?</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>700018</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addi $t0, $0, 1</td>
</tr>
<tr>
<td>00400024</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>?</td>
<td>700018</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>addu $t1, $0, $0</td>
</tr>
<tr>
<td>00400028</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>700018</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bge $t1,$a0,w2</td>
</tr>
<tr>
<td>00400038</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>700018</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>add $v0,$0,$t0</td>
</tr>
<tr>
<td>0040003c</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>700018</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>jr $ra</td>
</tr>
<tr>
<td>00700018</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>700018</td>
</tr>
</tbody>
</table>
**Von Neuman & Harvard CPU Architectures**

**Von Neuman architecture**
Area efficient but requires higher bus bandwidth because instructions and data must compete for memory.

**Harvard architecture** was coined to describe machines with separate memories. **Speed efficient**: Increased parallelism.
Multi-cycle Processor Datapath

- **Mux 0/1**: Multiplexer for selecting signals.
- **aluOp**: ALU operation.
- **aluSrcA**: ALU source A.
- **aluSrcB**: ALU source B.
- **aluControl**: ALU control signals.
- **aluResult**: ALU result.
- **aluOut**: ALU output.
- **irWrite**: Instruction register write.
- **regDst**: Register destination.
- **regWrite**: Register write.
- **memRead**: Memory read.
- **memWrite**: Memory write.
- **memtoReg**: Memory to register.
- **iorD**: Instruction operation.
- **memData**: Memory data.
- **memAddress**: Memory address.
- **pc**: Program counter.

The diagram illustrates the flow of data and control signals through the multiplexer and ALU components of the processor's datapath.
Multi-cycle Datapath: with controller
Multi-cycle using Finite State Machine

Finite State Machine
( *hardwired control*)

Combination control logic

Datapath control outputs

Inputs

State register

Inputs from instruction register opcode field

Next state
Finite State Machine: program overview

1. Fetch
2. Decode
3. Rformat1
4. BEQ1
5. JUMP1
6. Mem1
7. LW2
8. SW2
9. LW2+1
The Four Stages of R-Format

- **Fetch:**
  - Fetch the instruction from the Instruction Memory

- **Decode:**
  - Registers Fetch and Instruction Decode

- **Exec: ALU**
  - ALU operates on the two register operands
  - Update PC

- **Write: Reg**
  - Write the ALU output back to the register file
R-Format State Machine

Clock=1

Cycle 1
Cycle 2
Cycle 3
Cycle 4

R-Format | Ifetch | Reg/Dec | Exec | Wr |

Clock=1

Decode → Clock=1

Fetch

Clock=1

Exec

Write

Clock=1

CWRU EECS 318
The Five **Stages** of Load Instruction

- **Fetch:**
  - Fetch the instruction from the Instruction Memory

- **Decode:**
  - Registers Fetch and Instruction Decode

- **Exec:** Offset
  - Calculate the memory offset

- **Mem:**
  - Read the data from the Data Memory

- **Wr:**
  - Write the data back to the register file
R-Format & I-Format State Machine

Decode

Fetch

Need to check instruction format

Clock=1

Clock=1 AND R-Format=1

Clock=1 AND I-Format=1

Clock=1

Offset

Exec

ALU

Write

Reg

Write Reg

Reg

Mem

Read

Need to check opcode

Write Reg

Need to check opcode

Clock=1 AND opcode=LW
Multi-Instruction sequence
State machine stepping: $T_1$ Fetch

(Done in parallel) IR $\leftrightarrow$ MEMORY[PC] & PC $\leftarrow$ PC + 4
### T₁ Fetch: State machine

**Cycle 1**
- **MemRead**: 1
- **MemWrite**: 0
- **IorD**: 1 \((\text{MemAddr} \leftarrow \text{PC})\)
- **IRWrite**: 1 \((\text{IR} \leftarrow \text{Mem[PC]})\)
- **ALUSrcA**: 0 \(=\text{PC}\)
- **ALUSrcB**: 1 \(=4\)
- **ALUOP**: ADD \((\text{PC} \leftarrow 4+\text{PC})\)
- **PCWrite**: 1, **PCSource**: 1 \(=\text{ALU}\)
- **RegWrite**: 0, **MemtoReg**: \(X\), **RegDst**: \(X\)

**Cycle 2**
- **MemRead**: 1
- **MemWrite**: 0
- **IorD**: 1 \((\text{MemAddr} \leftarrow \text{PC})\)
- **IRWrite**: 1 \((\text{IR} \leftarrow \text{Mem[PC]})\)
- **ALUSrcA**: 0 \(=\text{PC}\)
- **ALUSrcB**: 1 \(=4\)
- **ALUOP**: ADD \((\text{PC} \leftarrow 4+\text{PC})\)
- **PCWrite**: 1, **PCSource**: 1 \(=\text{ALU}\)
- **RegWrite**: 0, **MemtoReg**: \(X\), **RegDst**: \(X\)

### Instruction Fetch

- Start
- R-Format
- Ifetch
- Reg/Dec
- Exec
- Wr

**Decode** → **Exec** → **Write Reg**
T_2 Decode (read $rs and $rt and offset+pc)

A ← Reg[IR[25-21]] & B ← Reg[IR[20-16]]
& ALUOut ← PC + signext(IR[15-0]) << 2
MemRead=0, MemWrite=0
IorD=X
IRWrite=0
ALUSrcA=0 (=PC)
ALUSrcB=3 (=signext(IR<<2))
ALUOP=0 (=add)
PCWrite=0, PCSource=X
RegWrite=0,
MemtoReg=X, RegDst=X
Instr. Decode & Register Fetch
$T_3$ ExecALU (ALU instruction)

ALUOut ← A op(IR[31-26]) B
R-Format Execution

MemRead=0, MemWrite=0
IorD=X
IRWrite=0
ALUSrcA=1 (=A =Reg[$rs])
ALUSrcB=0 (=B =Reg[$rt])
ALUOP=2 (=IR[28-26])
PCWrite=0, PCSource=X
RegWrite=0,
MemtoReg=X, RegDst=X
T_4 WrReg (ALU instruction)

T4 WrReg State machine

Start

R-Format  Ifetch  Reg/Dec  Exec  Wr

Fetch → Decode

Decode → Exec

R-Format Write Register

MemRead=0, MemWrite=0
IorD=X
IRWrite=0
ALUSrcA=X
ALUSrcB=X
ALUOP=X
PCWrite=0, PCSource=X
RegWrite=1, (Reg[$rd] ← ALUout)
MemtoReg=0, (=ALUout)
RegDst=1, (=rd)
Moore Machines

- So far we considered Moore machines where the output $O$ is a function of only the current state $Q$.

- Moore FSM State Transition Graph
# Moore Output State Tables: O(State)

<table>
<thead>
<tr>
<th>State</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$-R</th>
<th>$T_4$-R</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemRead</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MUX IorD</td>
<td>0 =PC</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IRWrite</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ALUOP</td>
<td>0 +</td>
<td>0 +</td>
<td>2 =op</td>
<td>X</td>
</tr>
<tr>
<td>MUX ALUSrcA</td>
<td>0 =PC</td>
<td>0 =PC</td>
<td>1 =A =$rs$</td>
<td>X</td>
</tr>
<tr>
<td>MUX ALUSrcB</td>
<td>1 =4</td>
<td>3 =offset</td>
<td>0 =B =$rt$</td>
<td>X</td>
</tr>
<tr>
<td>PCWrite</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MUX PCSource</td>
<td>0 =ALU</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RegWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MUX MemtoReg</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0 =ALUOut</td>
</tr>
<tr>
<td>MUX RegDst</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1 =$rd$</td>
</tr>
</tbody>
</table>
**Review: The Five Stages of Load Instruction**

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
</tr>
</tbody>
</table>

- **Fetch:**
  - Fetch the instruction from the Instruction Memory

- **Decode:**
  - Registers Fetch and Instruction Decode

- **Exec:** Offset
  - Calculate the memory offset

- **Mem:**
  - Read the data from the Data Memory

- **Wr:**
  - Write the data back to the register file
Need to check instruction format

Clock=1

Decode

Clock=1 AND
R-Format=1

Exec
ALU

Clock=1 AND
I-Format=1

Mem
Read

Clock=1 AND
opcode=LW

Need to check
opcode

Write
Reg

Clock=1

Write
Reg

Clock=1

Fetch

Clock=1

Exec
Offset

Need to check
instruction format
$T_3$–I $\text{Mem1} \ (\text{common to both load \& store})$

$\text{ALUOut} \leftarrow A + \text{sign\_extend}(\text{IR}[15-0])$
Mem1 I-Format State Machine \(=rs + \text{offset}\)
\( T_4 \) – LW1: load instruction, read memory

MDR \leftrightarrow \text{Memory}[\text{ALUOut}]
T4 LW2 I-Format State Machine = Mem[ALU]

- **Decode**
  - Clock=1 AND R-Format=1

- **Exec Offset**
  - Clock=1 AND I-Format=1

- **I-Format Memory Read**

  - MemRead=1, MemWrite=0
  - IorD=1
  - IRWrite=0
  - ALUOP=X
  - ALUSrcA=X
  - ALUSrcB=X
  - PCWrite=0, PCSource=X
  - RegWrite=0,
  - MemtoReg=X, RegDst=X

- **Fetch**

- **Write Reg**
  - Clock=1 AND opcode=LW

- **Exec ALU**

\( T_5 \) - LW2 Load instruction, write to register

\[ \text{Reg[ IR[20-16] ]} \leftarrow \text{MDR} \]
T5 LW2 I-Format State Machine $rt=MDR

- **Decode**
  - Clock=1 AND R-Format=1

- **Exec ALU**
  - Clock=1 AND opcode=LW
  - MemRead=1, MemWrite=0
  - IorD=1
  - IRWrite=0
  - ALUOP=X
  - ALUSrcA=X
  - ALUSrcB=X
  - PCWrite=0, PCSource=X
  - RegWrite=1,
  - MemtoReg=1, RegDst=1

- **Exec Offset**
  - Clock=1 AND I-Format=1

- **Mem Read**
  - Clock=1 AND opcode=LW

- **Write Reg**
  - Clock=1 AND I-Format=1
T₄—SW2 Store instruction, write to memory

Memory[ ALUOut ] ← B
CWRU EECS 318

T₄ SW2 I-Format State Machine Mem[ALU] =

- **Decode**
  - Clock = 1 AND R-Format = 1

- **Exec**
  - Clock = 1 AND I-Format = 1

- **ALU**
  - MemRead = 0, MemWrite = 1
  - IorD = 1
  - IRWrite = 0
  - ALUOP = X
  - ALUSrcA = X
  - ALUSrcB = X
  - PCWrite = 0, PCSource = X
  - RegWrite = 0
  - MemtoReg = X, RegDst = X

- **Offset**
  - Clock = 1 AND opcode = SW

- **Mem Read**
  - Store not Load!
T₃ BEQ1 (Conditional branch instruction)

If (A - B == 0) { PC ← ALUOut; }

ALUOut = Address computed in T₂!
T3 BEQ1 I-Format State Machine = $rs + offset

Decide: Clock = 1 AND R-Format = 1

Fetch

Decode

钟=1 AND opcode=branch

Exec

ALU

Write Reg

MemRead = 0, MemWrite = 0
IorD = X
IRWrite = 0
ALUOP = 0 = subtract
ALUSrcA = 1 = A = Reg[$rs]
ALUSrcB = 0 = B = Reg[$rt]
PCWrite = 0,
PCWriteCond = 1,
PCSource = 1 = ALUout
RegWrite = 0,
MemtoReg = X, RegDst = X

B-Format Execution
$T_3$ Jump1 (Jump Address)

$\text{PC} \leftarrow \text{PC}[31-28] \ || \ \text{IR}[25-0] \ll 2$
## Moore Output State Tables: O(State)

<table>
<thead>
<tr>
<th>State</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃-R</th>
<th>T₄-R</th>
<th>T₃-I</th>
<th>T₄-SW</th>
<th>T₄-LW</th>
<th>T₅-LW</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemRead</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MUX IorD</td>
<td>0=PC</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1=ALU</td>
<td>1=ALU</td>
<td>X</td>
</tr>
<tr>
<td>IRWrite</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ALUOP</td>
<td>0=+</td>
<td>0</td>
<td>2=op</td>
<td>X</td>
<td>0=add</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUX ALUSrcA</td>
<td>0=PC</td>
<td>0</td>
<td>1=A=$rs</td>
<td>X</td>
<td>1=A=$rs</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MUX ALUSrcB</td>
<td>1=4</td>
<td>3</td>
<td>0=B=$rt</td>
<td>X</td>
<td>2=sign</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PCWrite</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MUX PCSource</td>
<td>0=AL</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>RegWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MUX MemtoReg</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0=ALU</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1=Mdr</td>
</tr>
<tr>
<td>MUX RegDst</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1=$rd</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1=$rt</td>
</tr>
</tbody>
</table>
Multi-cycle: 5 execution steps

- $T_1 (a, lw, sw, beq, j)$ Instruction Fetch
- $T_2 (a, lw, sw, beq, j)$ Instruction Decode and Register Fetch
- $T_3 (a, lw, sw, beq, j)$ Execution, Memory Address Calculation, or Branch Completion
- $T_4 (a, lw, sw)$ Memory Access or R-type instruction completion
- $T_5 (a, lw)$ Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!
### Multi-cycle Approach

All operations in each clock cycle $T_i$ are done in parallel not sequential!

For example, $T_1$, IR = Memory[PC] and PC=PC+4 are done simultaneously!

<table>
<thead>
<tr>
<th>Step name</th>
<th>Action for R-type instructions</th>
<th>Action for memory-reference instructions</th>
<th>Action for branches</th>
<th>Action for jumps</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$ Instruction fetch</td>
<td></td>
<td>IR = Memory[PC]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC = PC + 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_2$ Instruction decode/register fetch</td>
<td></td>
<td>A = Reg [IR[25-21]]</td>
<td>B = Reg [IR[20-16]]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALUOut = PC + (sign-extend (IR[15-0]) &lt;&lt; 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_3$ Execution, address computation, branch/jump completion</td>
<td>ALUOut = A op B</td>
<td>ALUOut = A + sign-extend (IR[15-0])</td>
<td>if (A == B) then PC = ALUOut</td>
<td>PC = PC [31-28] Il (IR[25-0]&lt;&lt;2)</td>
</tr>
<tr>
<td>$T_4$ Memory access or R-type completion</td>
<td>Reg [IR[15-11]] = ALUOut</td>
<td>Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_5$ Memory read completion</td>
<td>Load: Reg[IR[20-16]] = MDR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Between Clock $T_2$ and $T_3$ the microcode sequencer will do a dispatch 1