Tema 2: Disseny del processador escalar: datapath i unitat de control

Eduard Ayguadé i Josep Llosa
Chapter outline

- RISC: Reduced Instruction Set Computer
  - An example: MIPS
- Goals in processor implementation
- Brief review of sequential logic design
- A simple implementation of a MIPS integer instruction subset
- A multi-cycle MIPS implementation
- Exceptions
RISC has its roots in
- The IBM 801 project started by John Cocke at IBM Research in 1975
  ✓ Precursor to the IBM RS/6000 workstation processors which later influenced PowerPC
- The Berkeley RISC project started by D. Patterson in 1980
  ✓ Evolved into the SPARC ISA of Sun Microsystems
- The Stanford MIPS project started by J. Hennessy ~1980
  ✓ Hennessy co-founded MIPS Computer

RISC philosophy is that instruction sets should be simplified to enable fast hardware implementations that can be exploited by optimizing compilers
Original RISC approach

- **Fixed-length (32 bits for MIPS) instructions that have only a few formats**
  - Simplifies instruction fetch and decode
  - Sacrifices code density
    - ✓ Some bits are wasted for some instruction types
    - ✓ Requires more memory

- **Register-register (load-store) architecture**
  - Allows for very fast implementation of simple instructions
  - Easier to pipeline (see chapter 4)
  - Sacrifices code density: more instructions than register-memory and memory-memory ISA (Instruction Set Architecture)
Original RISC approach

- Limited number of addressing modes
  - Simplifies effective address calculation and speeds up memory access

- Few if any complex arithmetic functions
  - Instead use more, simpler instructions
Some recent deviations from RISC

- Shorter instructions (16 bits) intermixed with longer (32 bits)
  - Higher code density for embedded environments

- Some complex operations
  - Multiply-add for DSP and multimedia applications

- Experience is that selectively adding some complexity back into the ISA improves performance and/or cost effectiveness
MIPS instruction formats

- **R-type (Register)**
  
  
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td></td>
<td>shamt</td>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **I-type (Immediate)**
  
  
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **J-Type (Jump)**
  
  
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td></td>
<td>target</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MIPS instruction formats

... where:
- op is a 6-bit operation code (opcode)
- rs is a 5-bit source register specifier
- rt is a 5-bit (source or destination) register specifier or branch condition
- rd is a 5-bit destination register specifier
- shamt is a 5-bit shift amount
- funct is a 6-bit function field
- immediate is a 16-bit immediate, branch displacement, or memory address displacement
- target is a 26-bit jump target address

Simplifications
- Fixed length
- Limited number of field types
- Many fields located in same location in different formats
## MIPS instruction summary

### MIPS operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>2&lt;sup&gt;30&lt;/sup&gt; memory words</td>
<td>Memory[0], Memory[4], ..., Memory[4294967292]</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

### MIPS assembly language

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>addi $s1, $s2, 100</td>
<td>$s1 = $s2 + 100</td>
<td>Used to add constants</td>
</tr>
<tr>
<td>Data transfer</td>
<td>lw</td>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Word from register to memory</td>
</tr>
<tr>
<td></td>
<td>lb</td>
<td>lb $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td></td>
<td>sb</td>
<td>sb $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Byte from register to memory</td>
</tr>
<tr>
<td></td>
<td>lui</td>
<td>lui $s1, 100</td>
<td>$s1 = 100 * 2&lt;sup&gt;16&lt;/sup&gt;</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>beq</td>
<td>beq $s1, $s2, 25</td>
<td>if ($s1 == $s2) go to PC + 4 + 100</td>
<td>Equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td>bne $s1, $s2, 25</td>
<td>if ($s1 != $s2) go to PC + 4 + 100</td>
<td>Not equal test; PC-relative</td>
</tr>
<tr>
<td></td>
<td>slt</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than; for beq, bne</td>
</tr>
<tr>
<td></td>
<td>slti</td>
<td>slti $s1, $s2, 100</td>
<td>if ($s2 &lt; 100) $s1 = 1; else $s1 = 0</td>
<td>Compare less than constant</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>j</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>Jump to target address</td>
</tr>
<tr>
<td></td>
<td>jr</td>
<td>jr $ra</td>
<td>go to $ra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jal</td>
<td>jal 2500</td>
<td>$ra = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
MIPS addressing modes

1. Immediate addressing
   - op  rs  rt  Immediate

2. Register addressing
   - op  rs  rt  rd  ...  funct
   - Registers
     - Register

3. Base addressing
   - op  rs  rt  Address
   - Register
     - Memory
       - Byte
       - Halfword
       - Word

4. PC-relative addressing
   - op  rs  rt  Address
   - PC
     - Memory
       - Word

5. Pseudodirect addressing
   - op  Address
   - PC
     - Memory
       - Word
Registers in MIPS

32 GPRs
- $0 is hardwired to zero in all MIPS implementations
  - Load a constant, e.g., addi $2,$0,#3
  - Absolute addressing mode, e.g., lw $4,1000($0)
- $31 holds return address for subroutine returns
- All other GPRs are “generic” although software conventions may define their usage

- Program counter register
  - HI
  - LO
  - PC
All operands and addresses are 32 bits

Instructions:

- Arithmetic and logic: add, sub, and, or, slt
  \[ \text{op } R_d, R_s, R_t \rightarrow R_d = R_s \text{ op } R_t \quad \text{op} = \{+, -, \text{and, or}\} \]
  \[ \text{slt } R_d, R_s, R_t \rightarrow R_d = 1 \text{ if } (R_s < R_t); \text{ otherwise } R_d = 0 \]

- Memory access: ld, st
  \[ \text{ld } R_t, R_s, \text{ desp} \rightarrow R_t = \text{mem}[R_s + \text{ desp}] \]
  \[ \text{st } R_t, R_s, \text{ desp} \rightarrow \text{mem}[R_s + \text{ desp}] = R_t \]

- Control flow change: beq, j
  \[ \text{beq } R_s, R_t, \text{ desp} \rightarrow \text{if } (R_t = R_s) \text{ then } \text{PC} = \text{PC} + 4 + \text{ desp} \]
  \[ \text{j desp} \rightarrow \text{PC} = \text{ desp} \]
Balance the rate of *supply* of instructions and data and the rate at which the execution core can *consume* them and can update memory.
Our processor design progression

\[ T_{\text{exec}} = N \times \text{CPI} \times t_c \]

- (1) Instruction fetch, execute, and operand reads from data memory all occur in the same clock cycle
  
  \[ \text{CPI minimum (CPI}=1) \]

- (2) Instruction fetch, execute, and operand reads from data memory occur in separate clock cycles
  
  \[ t_c \text{ minimum, CPI} > 1 \]

- (3) A pipelined design (Chapter 4)

- (4) A superscalar design: \( W > 1 \) (Chapter 5)
State elements are clocked devices
- PC register, register file and other intermediate registers

Combinatorial elements hold no state
- ALU, caches, multiplier, multiplexers, etc.

In edge triggered clocking, state elements are only updated on the (rising) edge of the clock pulse
Brief review of sequential logic design

- The same state element can be read at the beginning of a clock cycle and updated at the end

Example: incrementing the PC
- Memory to hold instructions
- Register to hold the instruction memory address
- Logic to generate the next instruction address (PC+4 points to next sequential instruction in memory)
Execution datapath

- Only look at MIPS subset
  - add, sub, and, or
  - lw, sw
  - slt
  - beq, j

- For all instructions except j, we
  - Read operands from the register file
  - Perform an ALU operation

- For all instructions except sw, beq, and j, we write a result into the register file
- Read register 1,2: source operand register numbers
- Read data 1,2: source operands (32 bits each)
- Write register: destination operand register number
- Write data: data to be written into register file
- RegWrite: when asserted, enables writing
### Execution datapath

**Datapath for R-type (add, sub, and, or, slt)**

- **R-type instruction format:**

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>shamt</td>
<td>funct</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Execution datapath

Datapath for beq instruction

- I-type instruction format:

```
<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Zero ALU output indicates if rs=rt (branch is taken/not taken)
Most general-purpose ISAs specify separate integer and floating point register files
- Operand representation formats differ
- Computation hardware differs

Result is a split of the execution core into integer and floating point sections
Integer and floating point computation

Motorola's PowerPC 604e™ RISC Microprocessor

- Integer register file and execution units
- flt pt register file and execution units
Data memory

- Used for lw, sw (I-type format)

- Address: memory location to be read or written
- Read data: data out of the memory on a load
- Write data: data into the memory on a store
- MemRead: indicates a read operation
- MemWrite: indicates a write operation
Datapath for lw, sw

- Address is the sign-extended immediate added to the source operand read out of the register file
- sw: data written to memory from specified register
- lw: data written to register file from memory address
3 MUXes
- Register file operand or sign extended immediate to ALU
- ALU or data memory output written to register file
- PC+4 or branch target address written to PC register
Example: datapath for R-type instructions

Example: add $4, $18, $30
Example: `lw $8, 112($2)`
Example: datapath for store instruction

Example: sw $10, 0($3)
Example: beq $28, $13, EXIT
Main control unit design
ALU operation control

- ALU control input codes

<table>
<thead>
<tr>
<th>ALU control input</th>
<th>ALU operation</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>and</td>
<td>and</td>
</tr>
<tr>
<td>001</td>
<td>or</td>
<td>or</td>
</tr>
<tr>
<td>010</td>
<td>add</td>
<td>add, lw, sw</td>
</tr>
<tr>
<td>110</td>
<td>subtract</td>
<td>sub, beq</td>
</tr>
<tr>
<td>111</td>
<td>set on less than</td>
<td>slt</td>
</tr>
</tbody>
</table>

- Two steps to generate the ALU control input
  - Use the opcode to distinguish R-type, lw and sw, and beq
  - If R-type, use funct field to determine the ALU control input
Opcode used to generate a 2-bit signal called ALUOp with the following encodings

- 00: lw or sw, perform an ALU add
- 01: beq, perform an ALU subtract
- 10: R-type, ALU operation is determined by the funct field

<table>
<thead>
<tr>
<th>Funct</th>
<th>Instruction</th>
<th>ALU control input</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>add</td>
<td>010</td>
</tr>
<tr>
<td>100010</td>
<td>sub</td>
<td>110</td>
</tr>
<tr>
<td>100100</td>
<td>and</td>
<td>000</td>
</tr>
<tr>
<td>100101</td>
<td>or</td>
<td>001</td>
</tr>
<tr>
<td>101010</td>
<td>slt</td>
<td>111</td>
</tr>
</tbody>
</table>
# Main control unit design

## Truth table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Memto-Reg</th>
<th>Reg Write</th>
<th>Mem Read</th>
<th>Mem Write</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format (0)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw (34)</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw (43)</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq (4)</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Adding support for jump instructions

- **J-type format**

  ![Diagram showing J-type format]

  Next PC formed by shifting left the 26-bit target two bits and combining it with the 4 high-order bits of PC+4

- **Now the next PC will be one of**
  - PC+4
  - beq target address
  - j target address

- **We need another MUX and control bit**
Adding support for jump instructions
All instructions take one clock cycle (CPI = 1)

Assume the following worst case delays:
- Instruction memory: 10 ns
- Data memory: 10 ns
- ALU: 10 ns
- Adders: 10 ns
- Register file: 5 ns
- MUXes, sign extension, gates, and shifters: 0 ns
Large disparity in worst case delays among instruction types:
- R-type: 30 ns
- beq: 25 ns
- j: 10 ns
- store: 35 ns
- load: 40 ns

Disparity would be worse in a real machine
- Even slower integer instructions (e.g., multiply/divide)
- Floating point instructions

$t_c = 40$ ns
## Evaluation of the simple implementation

### Slowdown w.r.t. variable $t_c$

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Probability</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>49%</td>
<td>30</td>
</tr>
<tr>
<td>load</td>
<td>22%</td>
<td>40</td>
</tr>
<tr>
<td>store</td>
<td>11%</td>
<td>35</td>
</tr>
<tr>
<td>beq</td>
<td>16%</td>
<td>25</td>
</tr>
<tr>
<td>j</td>
<td>2%</td>
<td>10</td>
</tr>
</tbody>
</table>

- Average $t_c = 0.49 \times 30 + 0.22 \times 40 + 0.11 \times 35 + 0.16 \times 25 + 0.02 \times 10 = 31.6$ ns
- Slowdown = $(N \times 1 \times 40) / (N \times 1 \times 31.6) = 1.27$
Different instruction types take different numbers of cycles to complete

<table>
<thead>
<tr>
<th>Step</th>
<th>R-type</th>
<th>lw, sw</th>
<th>beq</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td>IR=memory[PC]</td>
<td></td>
<td>PC=PC + 4</td>
<td></td>
</tr>
<tr>
<td>Decode/register fetch</td>
<td>A = RF[IR[25-21]]</td>
<td>B = RF[IR[20-16]]</td>
<td>Target=PC+(sign-extend(IR[15-0])&lt;&lt;2)</td>
<td>PC=Destination</td>
</tr>
<tr>
<td>Execution / @computation</td>
<td>ALU\text{output} = A \text{ op } B</td>
<td>ALU\text{output} = A + sign-extend(IR[15-0])</td>
<td>If (A==B) then PC=Target</td>
<td></td>
</tr>
<tr>
<td>Memory access</td>
<td>MDR=memory[ALU\text{output}] or memory[ALU\text{output}]=B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write register</td>
<td>RF[IR[20-16]]= ALU\text{output}</td>
<td>RF[IR[20-16]]=MDR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**A multicycle implementation**

- **Hardware can be shared**
  - One ALU for PC+4, branch target calculation, EA calculation and arithmetic operations
  - One memory for instructions and data

- **New registers store results of each step**
  - Not programmer visible!
Evaluation of the multicycle implementation

- Slowdown w.r.t. CPI=1

\[ t_c = 10 \text{ ns} \]
\[ \text{Average CPI} = 0.49 \times 4 + 0.22 \times 5 + 0.11 \times 4 + 0.16 \times 3 + 0.02 \times 2 = 4.02 \text{ cycles} \]
\[ \text{Slowdown} = \frac{(N \times 4.02 \times 10)}{(N \times 1 \times 40)} = 1.005 \]
Detailed multicycle datapath
Multicycle control unit
IR = Mem[PC]
Instruction fetch

- \( \text{PC} = \text{PC} + 4 \)
A=Reg[IR[25-21]], B=Reg[IR[20-16]]
**Instruction decode and register fetch**

- ALU\_output = PC + (sign-extend (IR[15-0]) << 2)
\[ \text{ALU}_{\text{output}} = A \, \text{op} \, B \]
Reg[IR[15-11]] = ALUOut
lw address computation cycle

\[ \text{ALUOut} = A + \text{sign-extend (IR[15-0])} \]
**MDR = Memory[ALUOut]**
Reg[IR[20-16]] = MDR
Control logic design

- Implemented as a Finite State Machine

- Inputs: 6 opcode bits
- Outputs: 16 control signals
- State: 4 bits for 10 states
Complete FSM

Instruction fetch
- MemRead
  - ALUSrcA = 0
  - IorD = 0
  - IWrite
  - ALUSrcB = 01
  - ALUOp = 00
  - PWrite
  - PCSource = 00

Instruction decode/register fetch
- ALUSrcA = 0
- ALUSrcB = 11
- ALUOp = 00

Start
- Memory address computation
  - ALUSrcA = 1
  - ALUSrcB = 10
  - ALUOp = 00

Execution
- (Op = "LW") or (Op = "SW")

Branch completion
- (Op = R-type)

Jump completion
- (Op = BGE)

Write-back step
- RegDst = 0
- RegWrite
- MemtoReg = 1

R-type completion
- RegDst = 1
- RegWrite
- MemtoReg = 0

Memory access
- Memory access
- Memory access

ALU operation
- ALU operation

PCWrite
- PCWrite
- PCSource = 10
An exception is an event that causes a deviation from the normal execution of instructions.

Exceptions may be generated internal to the processor or externally.

Types of exceptions:
- Arithmetic error (overflow, underflow, etc.)
- Undefined instruction
- Misaligned memory access (e.g., word access to odd address)
- Memory protection violation
- Page fault (request for instruction/data not in memory)
- Operating system call (e.g., read a file)
- Input/output device request
- Hardware error
- Power failure

We’ll only address overflow and undefined inst.
Steps required to handle exceptions

- Detect the exception
- Save enough information about the exception to handle it properly
- Save enough information about the program to resume it after the exception is handled
- Handle the exception
- Either terminate the program or resume executing it depending on the exception type
Detecting exceptions

overflow

undefined instruction
Saving exception information

- Performed by hardware
- We need to know what type of exception occurred and the PC of the instruction that had the exception
- In MIPS, the Cause register tells the type
  - Need an encoding for each exception type
  - Need a signal from the control unit to load it into the Cause register
- and the Exception Program Counter (EPC) register holds the PC
  - Need the control unit to subtract 4 from the PC register to get the correct PC (since we loaded PC+4 into the PC register during the Instruction Fetch cycle)
  - Need a signal from the control unit to load it into EPC
Saving exception information
Saving program information

- Performed by hardware and software
- EPC register holds the PC of the instruction that had the exception (where we will restart the program)
- The software routine that handles the exception (later) saves any registers that it will need to use to the stack and restores them when it is done
Handling the exception

- Performed by hardware and software
- Need to transfer control to a software routine to handle the exception (exception handler)
- The exception handler runs in a privileged mode that allows it to use special instructions and access all of memory
  - Our programs run in user mode
- The hardware enables the privileged mode, loads PC with the address of the exception handler, and transitions to the Fetch state
Handling the exception

- Loading the PC with exception handler address
### Exception handler steps

- Save away any registers it will use if the program is to be resumed
- Handles the particular exception
  - Operating system call: call subroutine associated with particular call
  - Underflow: set to value zero or use denormalized numbers
  - I/O: handle the particular I/O request, e.g., keyboard input
- Restore registers that were used if program is to be restarted
- Terminate the program, or resume execution by loading PC with EPC and transitioning to the Instruction Fetch state
FSM modifications

Instruction fetch

Start

MemRead
ALUSrcA = 0
forD = 0
ITWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00

Instruction decode/
Register fetch

ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00

Jump
completion

Jump
condition

Branch
completion

Op = BRI

Op = other

ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01

ALUSrcA = ALUSrcB = 00

ALUOp = 10

ALUSrcB = 00

PCWrite
PCSource = 10

RegDst = 1
RegWrite
MnemonicTrans = 0

MemWrite
forD = 1

MemRead
forD = 1

R-type completion

Memory access

Memory access

Write-back step

Overflow

Overflow

IntCause = 1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
EPCWrite
PCWrite
PCSource = 11

IntCause = 0
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11

(Op = `SW`) or (Op = `SW`)