ECE 596 – Master’s Project

Student: Alexander Mellas

Supervising Professor: Zuofu Cheng

Term: Fall 2024

ECE ILLINOIS

Table of Contents

RISC-V ISA Overview

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

RISC-V is a modern, open-standard Instruction Set Architecture (ISA) that is revolutionizing the design and implementation of processors

Simplicity of Design

From a hardware perspective, RISC-V offers several key advantages that make it appealing

- *RISC-V has a reduced instruction set, allowing for simpler hardware design*
- *The "modularity" of RISC-V allows designers to only implement the hardware relevant for desired instruction types*
- *Consistent encoding of instruction formats, simplifying decoding (e.g., rs1 is always located in bits 19-15)*

RISC-V Ecosystem & Adoption

The RISC-V ecosystem is rapidly expanding, with widespread adoption across industries and the development community

- *Adopted in sectors like IoT, automotive, and data centers*
- *Extensive ecosystem of tools, software, and dev platforms*
- *Backed by leading tech companies and academic institutions*
- *Continuous contributions from a global community*

RISC-V ISA Features

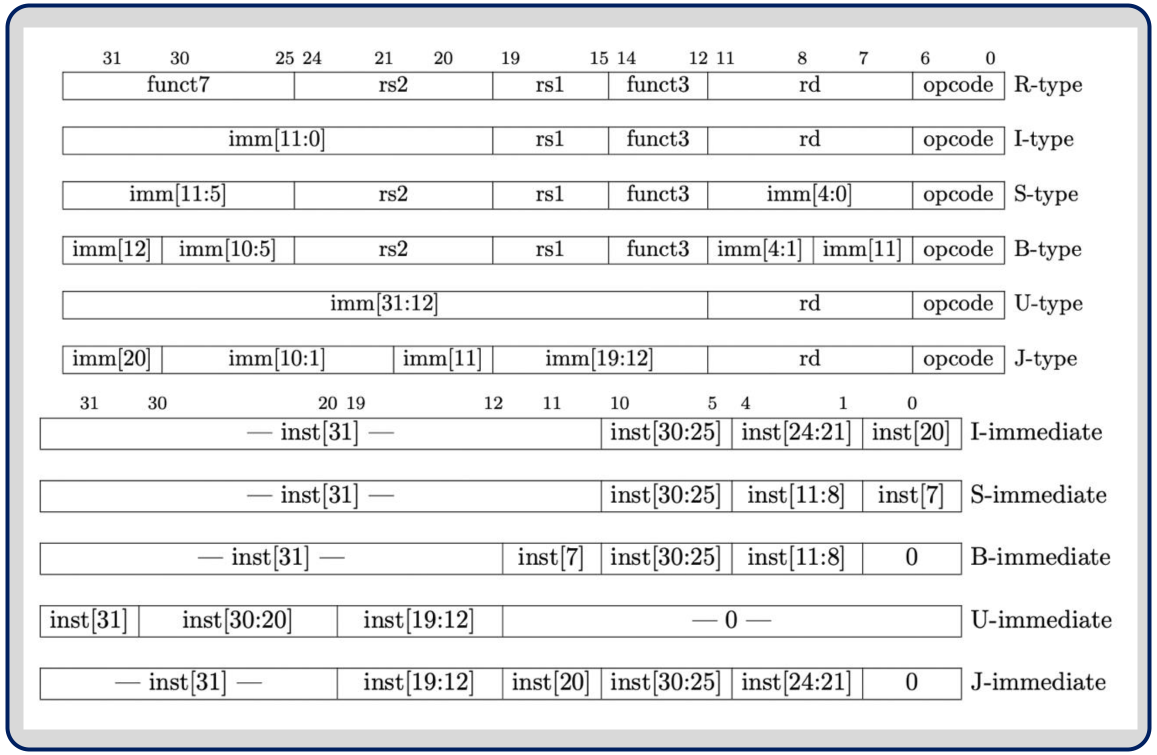
RISC-V provides a range of ISA configurations and extensions to meet diverse application needs

- *47 Base Instructions (37 excluding Fence, CSR, and ECALL/EBREAK)*
- *Base ISAs: RV32I, RV64I (32/64-bit)*
- *Extensions: M (Mult./Division), A (Atomic), F (Single-Precision Floating-Point), D (Double-Precision Floating-Point), C (Compressed)*
- *Specialized ISAs: RV32E, RV64E (Embedded 32/64-bit)*

RV32I Base Instruction Formats

The RV32I base instruction set uses six primary instruction formats, each designed for specific operations, enabling efficient and streamlined hardware implementation

R-Type (Register)	*Used for arithmetic and logical operations; it specifies two source registers and one destination register*
I-Type (Immediate)	*Supports operations involving immediate values, commonly used in load and arithmetic instructions*
S-Type (Store)	*Facilitates store operations with one source register for data and an immediate value for calculating the memory address*
B-Type (Branch)	*Manages conditional branching by comparing two registers and using an offset for the branch target address*
U-Type (Upper-Imm)	*Loads a 20-bit immediate value into the upper 20 bits of a register, useful for generating large constants or addresses*
J-Type (Jump)	*Supports unconditional jumps to a target address, using a 20-bit immediate value for the offset*

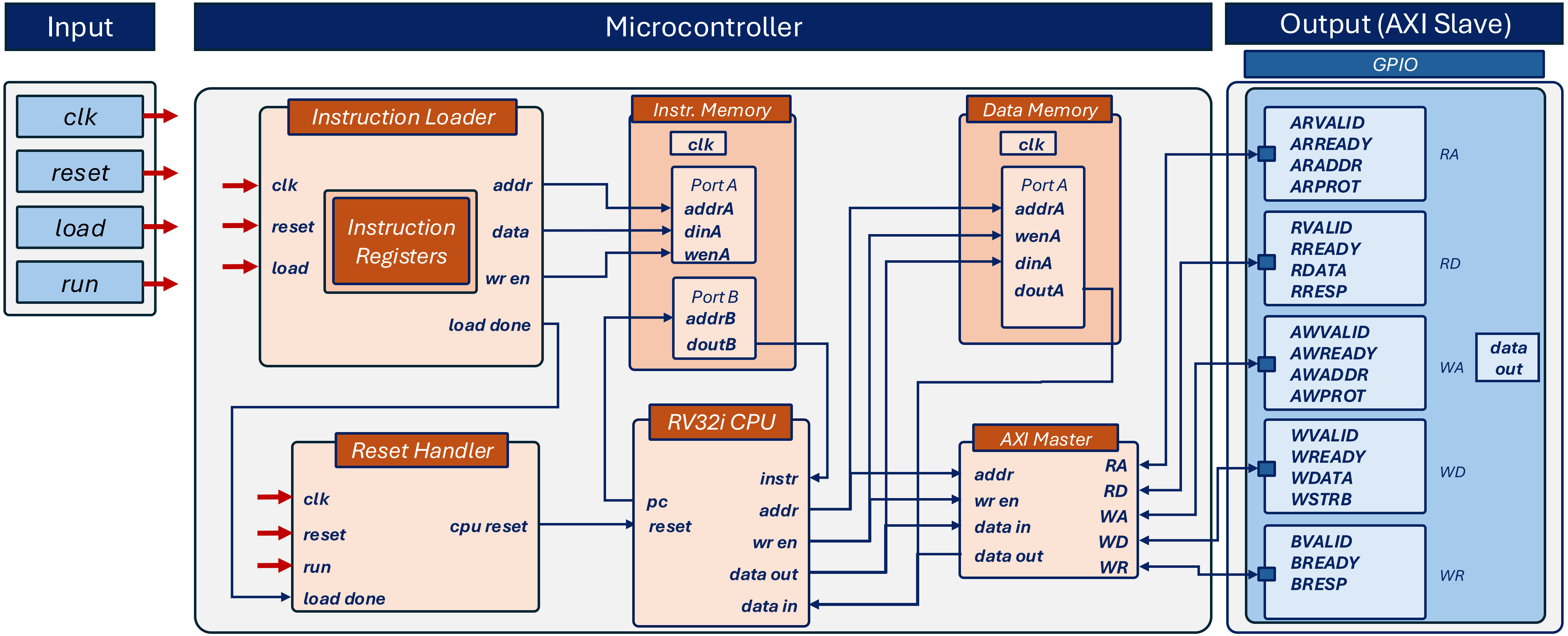

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type
31 30 20 19 12 11 10 5 4 1 0
— inst[31] — | inst[30:25] | inst[24:21] | inst[20] | I-immediate
— inst[31] — | inst[30:25] | inst[11:8] | inst[7] | S-immediate
— inst[31] — | inst[7] | inst[30:25] | inst[11:8] | 0 | B-immediate
inst[31] | inst[30:20] | inst[19:12] | — 0 — | U-immediate
— inst[31] — | inst[19:12] | inst[20] | inst[30:25] | inst[24:21] | 0 | J-immediate

RV32I Base Instruction Op-Types

Instruction	*Encoding*	*Description*
LUI	*U-Type*	• *Places the U-imm value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros*
AUIPC	*U-Type*	• *Forms a 32-bit offset from the 20-bit U-imm, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register rd*
JAL	*J-Type*	• *The J-imm is sign-extended and added to the PC to form the jump target address, and stores PC+4 in register rd*
JALR	*I-Type*	• *The target address is obtained by adding the 12-bit signed I-imm to rs1, then setting the least-significant bit of the result to zero; PC+4 is stored in rd*
Branch	*B-Type*	• *The 12-bit signed B-imm is added to the current PC to give the target address* • *Instructions compare two registers, and the branch is taken if the condition is true*
Load	*I-Type*	• *Loads copy a value from memory to register rd, where the effective byte address is obtained by adding rs1 to the sign-extended 12-bit I-imm*
Store	*S-Type*	• *Stores copy the value in register rs2 to memory, where the effective byte address is obtained by adding rs1 to the sign-extended 12-bit S-imm*
Reg-Imm ALU	*I-Type*	• *Performs a 32-bit ALU operation on a sign-extended I-imm and register rs1* • *The destination register is rd*
Reg-Reg ALU	*R-Type*	• *Performs a 32-bit ALU operation on registers rs1 and rs2* • *The destination register is rd*

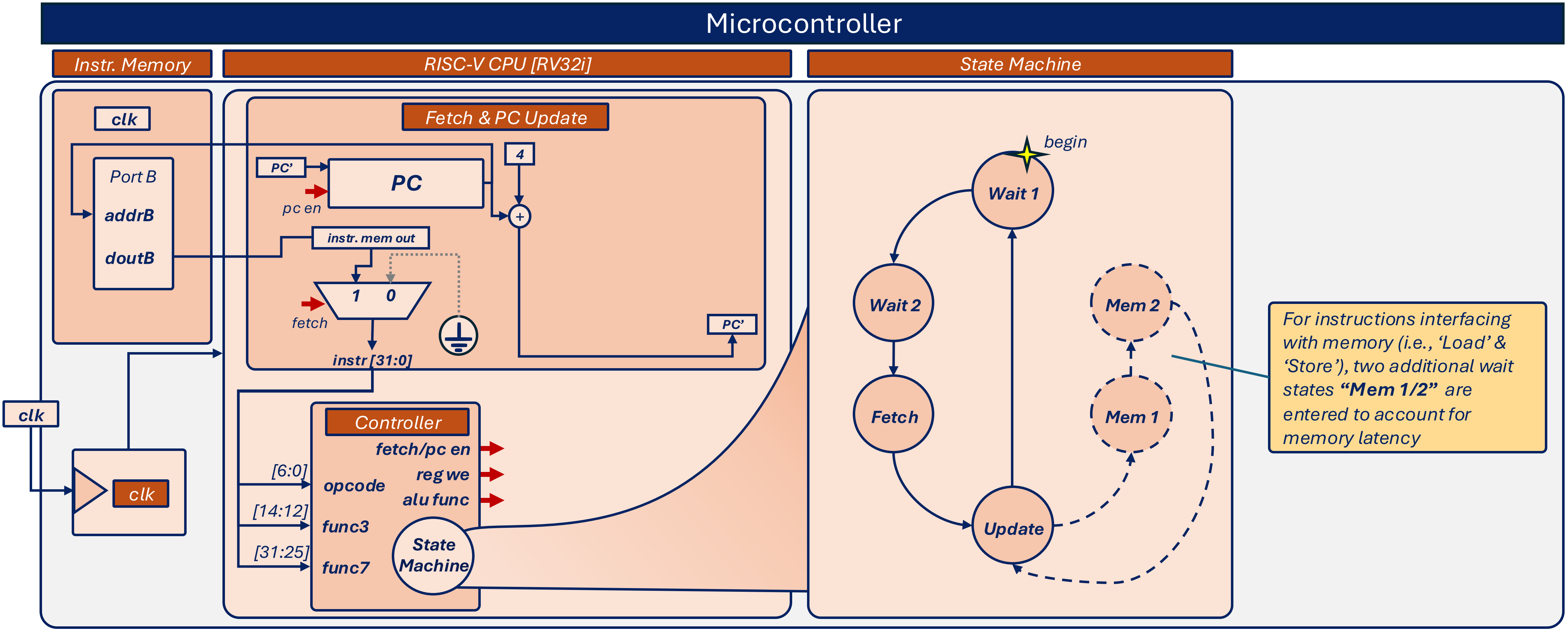
Table of Contents

System Top-Level



- *When "Load" is asserted, the 32-bit instructions are written to their corresponding indices of instruction memory, then "load done" is asserted*
- *With the assertion of "load done" and "run", the "Reset Handler" module de-asserts the "CPU Reset" to initiate program execution*
- *The CPU reads instructions from instruction memory, performs operations, and reads/writes from both data memory and via the AXI interface*
- *In this system, the Microcontroller is the AXI Master, and a GPIO module is the AXI Slave*

State Machine & Instruction Fetch



For instructions interfacing with memory (i.e., 'Load' & 'Store'), two additional wait states **"Mem 1/2"** are entered to account for memory latency

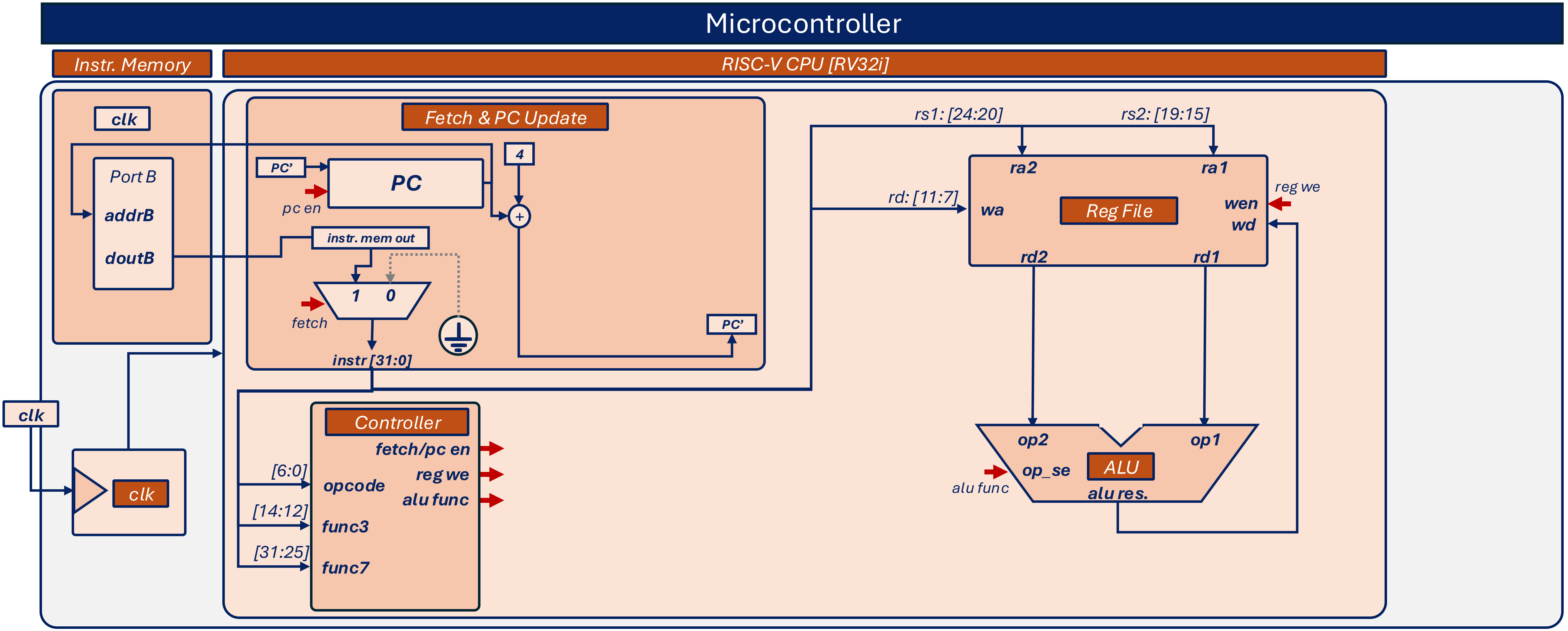
- *The output port of the program counter (PC) register is connected to the input address port of the instruction memory*
- *Two **"Wait"** states are allowed to account for the two clock-cycle delay in outputting the instruction corresponding to the input address*
- *During wait, "fetch" is de-asserted, grounding the instr. to avoid unintended executions, and once in the **"Fetch"** state, the "fetch" is asserted*
- *In the **"Update"** state, the instruction is decoded and "pc en" is asserted to update the PC register*

Arithmetic Operations

Integer computational instructions are either encoded as register-immediate operations using the I-type format or register-register using the R-type format

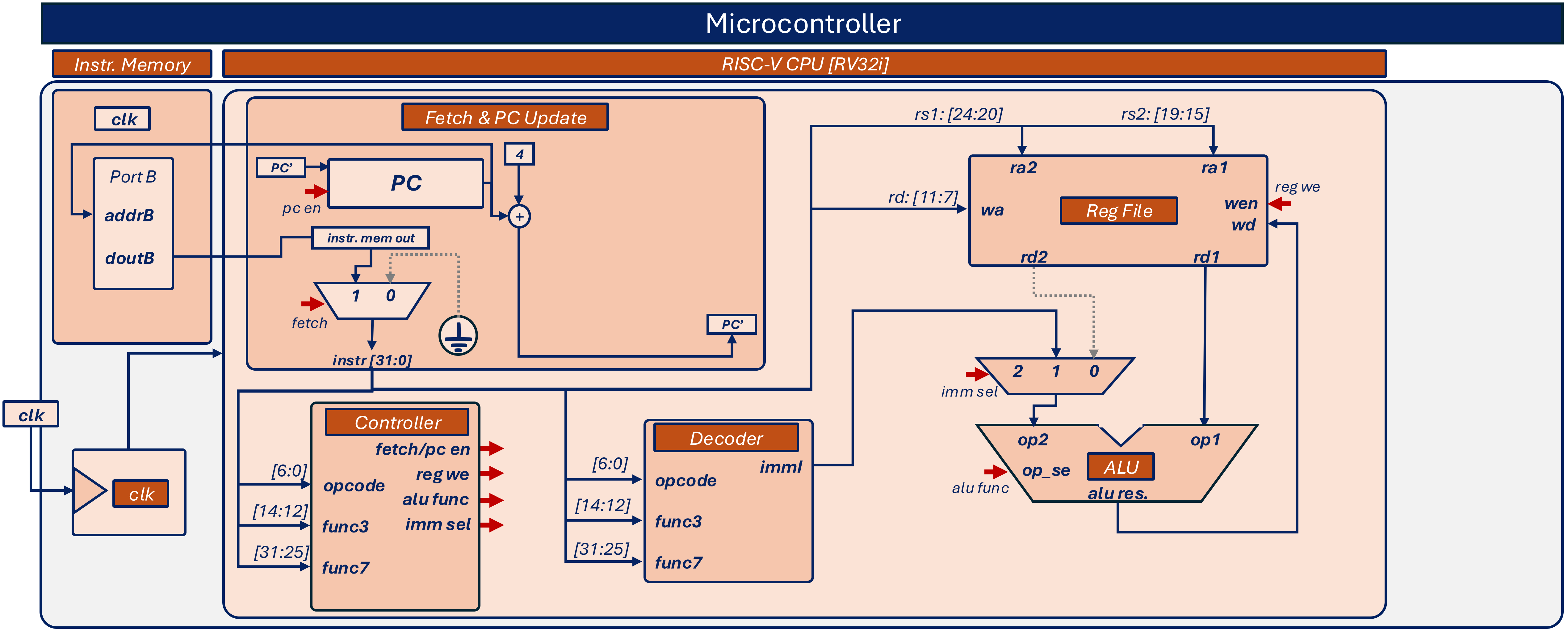
Operation	Description	funct7	funct3	Function
ADD / ADDI	Addition	0000000	000	c = a + b
SUB	Subtraction	0100000	000	c = a – b
SLT / SLTI	Set Less Than (Signed)	0000000	010	c = ($signed(a) < $signed(b)) ? 32'b1 : 32'b0
SLTU / SLTIU	Set Less Than (Unsigned)	0000000	011	c = (a < b) ? 32'b1 : 32'b0
XOR / XORI	Bitwise XOR	0000000	100	c = a ^ b
OR / ORI	Bitwise OR	0000000	110	c = a | b
AND / ANDI	Bitwise AND	0000000	111	c = a & b
SLL / SLLI	Shift Left Logical	0000000	001	c = a << b[4:0]
SRL / SRLI	Shift Right Logical	0000000	101	c = a >> b[4:0]
SRA / SRAI	Shift Right Arithmetic	0100000	101	c = $signed(a) >>> b[4:0]

“Integer Register-Register” Operations



- *The instruction fields rs1, rs2, and rd are connected directly to the corresponding ports of the “Register File”*
- *The register data outputs, rd1 and rd2, are the operands for the desired ALU operation*
- *The ALU function is conditioned on the func3 and func7 instruction fields, and the Controller sets the appropriate “alu func” signal*
- *Register write enable (“”reg we”) is asserted to allow writing to the reg file*

“Integer Register-Immediate” Operations



- *Here, the operands of the desired ALU operation are rd1 and imml (a sign-extended 12-bit offset)*
- *A multiplexor is added as there are now two possible sources for the second ALU operand (“op2”)*
- *The Controller produces a select signal (“imm sel”) to choose the appropriate input to op2, in this case imml*

Load & Store Operations

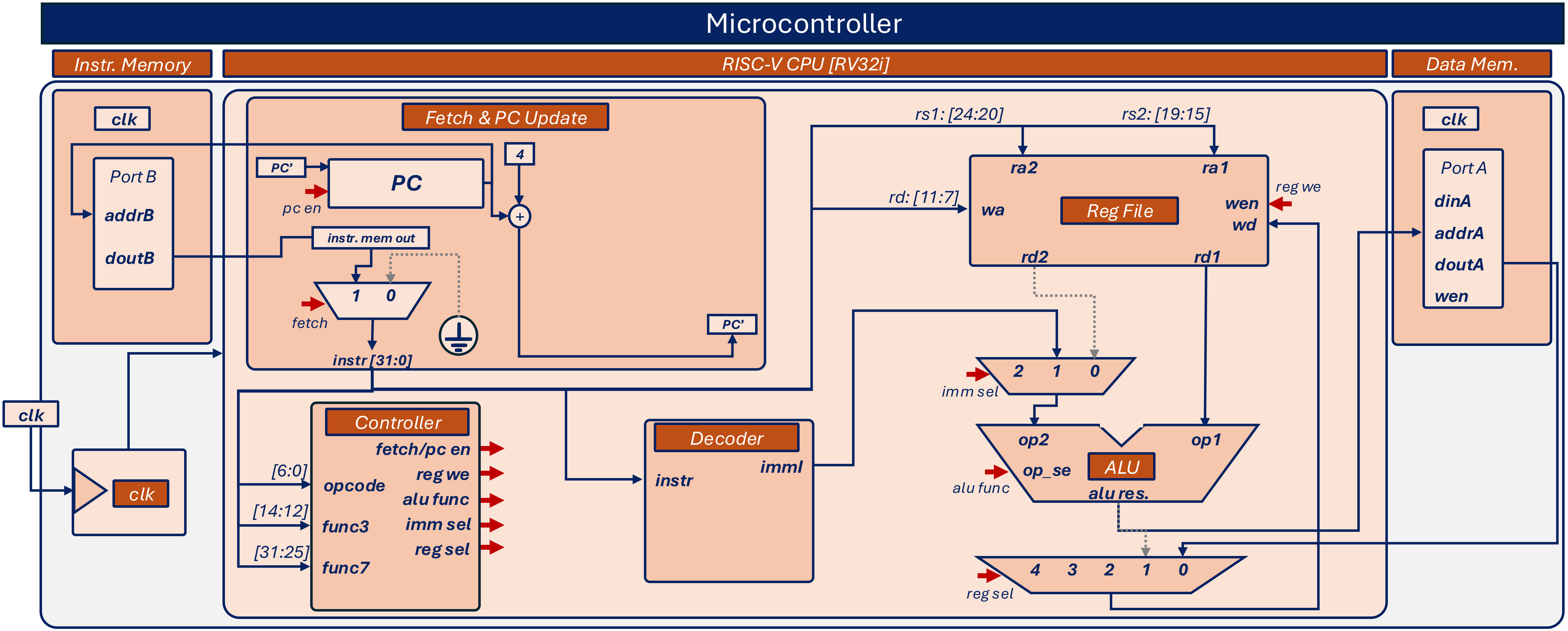
RV32I provides a 32-bit user address space that is byte-address and little-endian and data memory is accessed using the following instructions

logic [31:0] mem_rd_data → 32-bit word loaded from memory
logic [15:0] load_halfword → mem_rd_data[15:0]
logic [7:0] load_byte → mem_rd_data[7:0]

logic [31:0] reg_rd_data → 32-bit word from reg to be stored
logic [15:0] store_halfword → reg_rd_data[15:0]
logic [7:0] store_byte. → reg_rd_data[7:0]

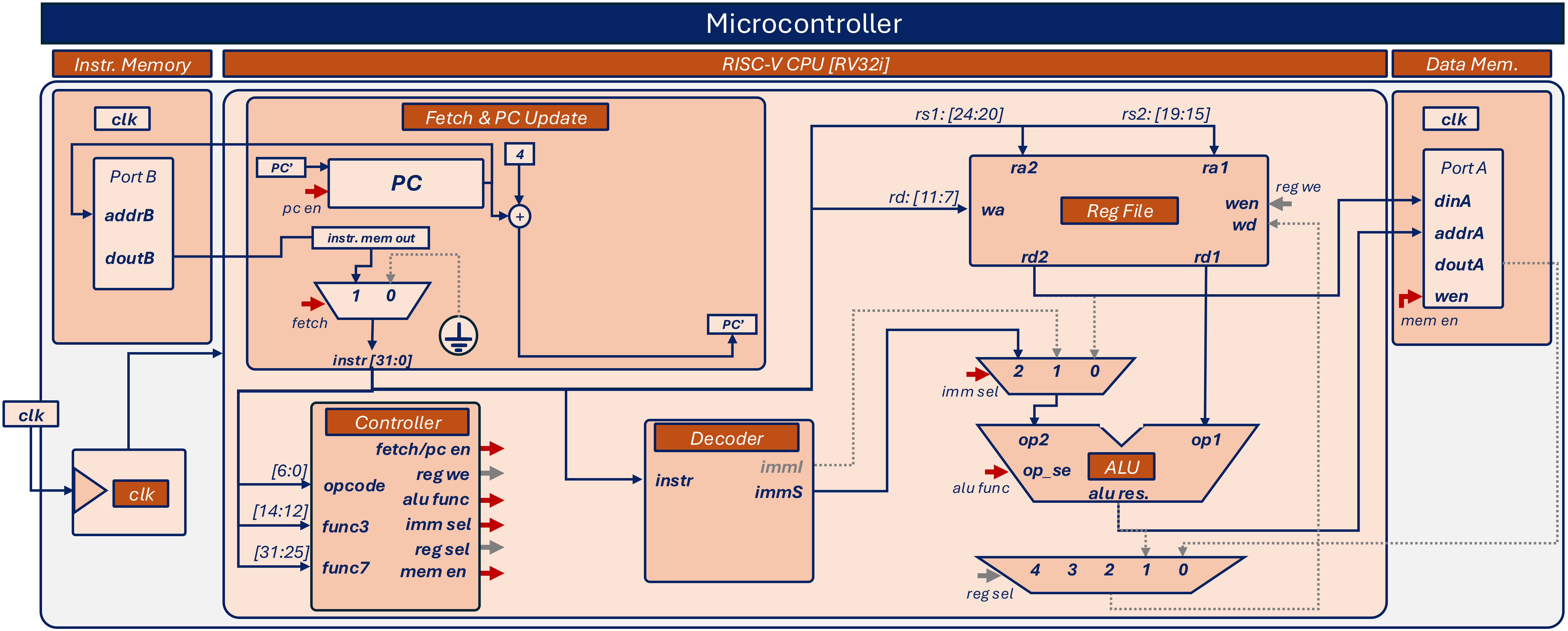
Operation	Description	funct3	Data
LW	Load Word	010	load_value = mem_rd_data
LH	Load Halfword	001	load_value = $signed(load_halfword)
LB	Load Byte	000	load_value = $signed(load_byte)
LBU	Load Byte Unsigned	100	load_value = {16’b0, load_halfword}
LHU	Load Halfword Unsigned	101	load_value = {24’b0, load_halfword}
SW	Store Word	010	store_value = reg_rd_data
SH	Store Halfword	001	store_value = {16’b0, store_halfword}
SB	Store Byte	000	store_value = {24’b0, store_byte}

“Load” Operations



- *The address to interface with data memory is produced by adding imml to rd1, and the ouput data is written to the register file*
- *As such, a multiplexor is added as there are now two possible sources for the write data port (“wd”) of the Register File*
- ***Note:** LW loads a 32-bit value from memory into rd; LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in rd; LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in rd; LB and LBU are defined analogously for 8-bit values*

“Store” Operations



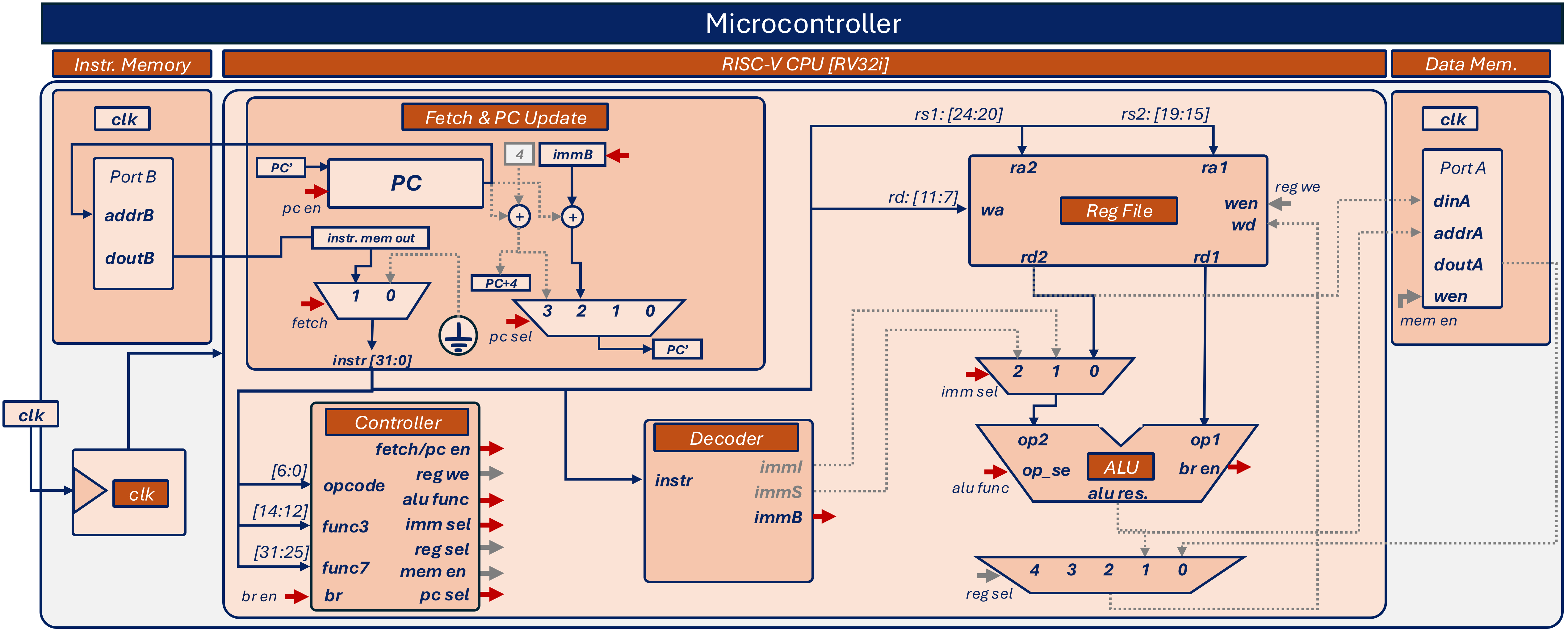
- *The data of rd2 is written to data memory at the address produced by adding rd1 to immS (12-bit S-offset)*
- *The decoder produces immS, which is connected to the mux that selects the ALU op2 input, and imm sel is appropriately updated*
- *The Control Logic asserts “mem en” to allow writing to memory, and de-asserts “reg we” such that no value is written to the Register File*
- ***Note:** The SW, SH, and SB instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register rs2 to memory*

Conditional Branch Operations

The Branch operations compare two operands, and if the condition is met, the output is true otherwise the output is false

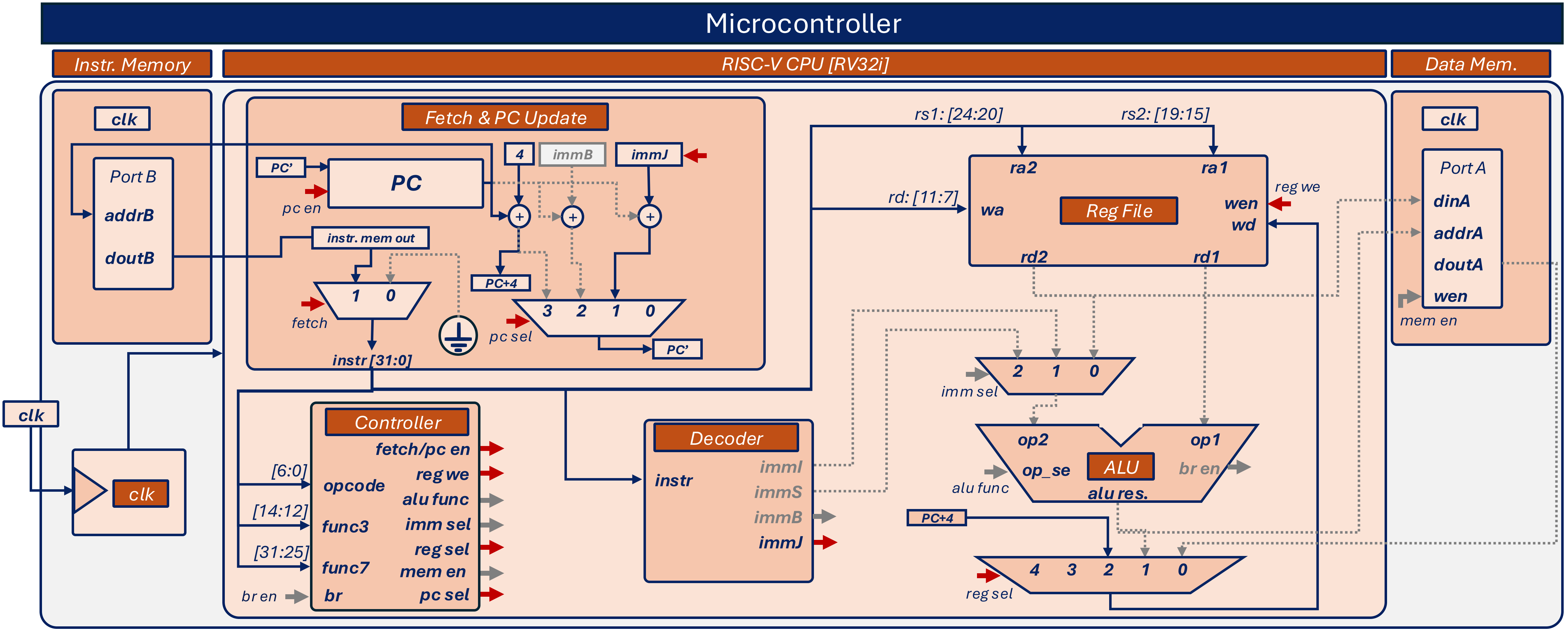
Operation	Description	funct3	Function
BEQ	Equals	000	c = (a == b) ? 1’b1 : 1’b0
BNE	Not Equals	001	c = (a != b) ? 1’b1 : 1’b0
BLT	Less Than (Signed)	100	c = ($signed(a) < $signed(b)) ? 1’b1 : 1’b0
BGE	Greater Than or Equal (Signed)	101	c = ($signed(a) >= $signed(b)) ? 1’b1 : 1’b0
BLTU	Less Than (Unsigned)	110	c = (a < b) ? 1’b1 : 1’b0
BGEU	Greater Than (Unsigned)	111	c = (a >= b) ? 1’b1 : 1’b0

"Branch" Operations



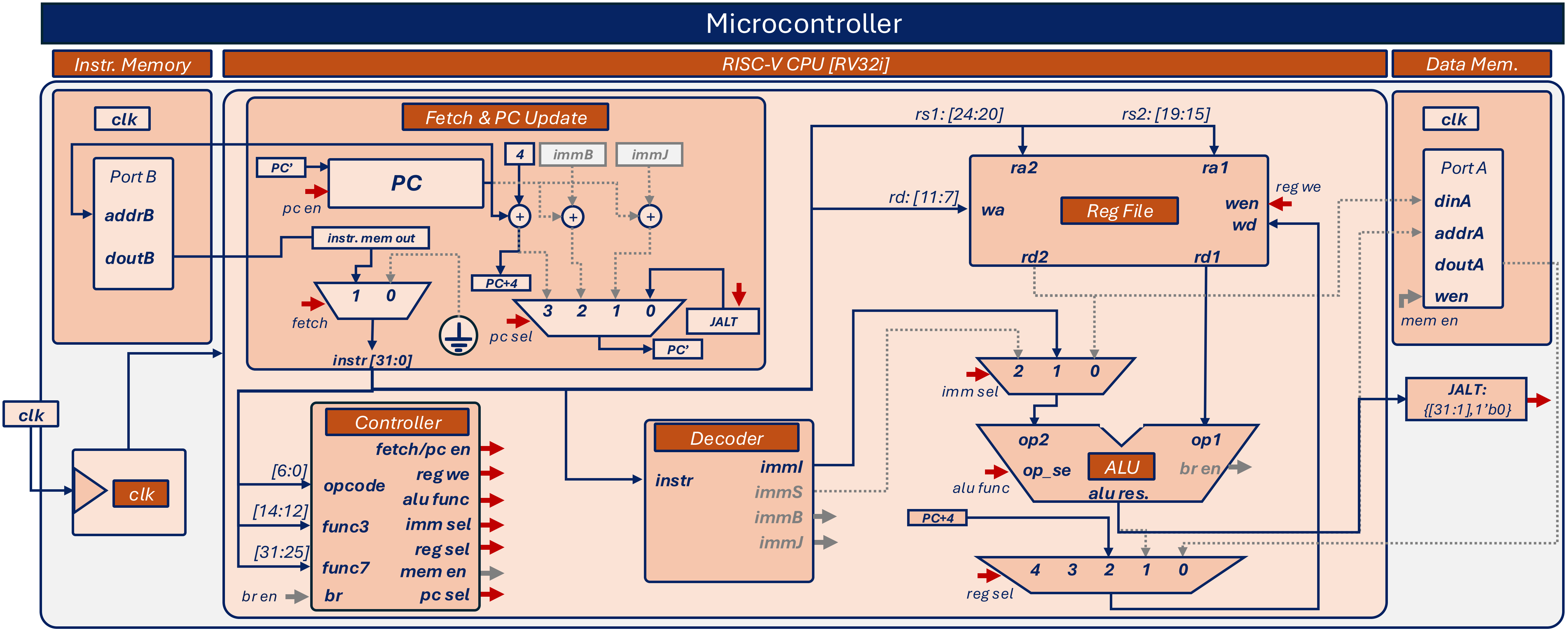
- *The decoder produces a 12-bit signed immB which is added to PC to give the target address*
- *Branch instructions compare rd1 and rd2, and if the ALU condition is met, the "br en" signal is asserted to indicate that the branch is taken*
- *A multiplexor is added to the "Fetch & PC Update Logic", as there are now two possible sources for the next PC value (" PC' ")*
- *The Control Logic produces a "pc sel" signal, the select signal for this mux, choosing the appropriate next PC value*

"Jump and Link" Operations



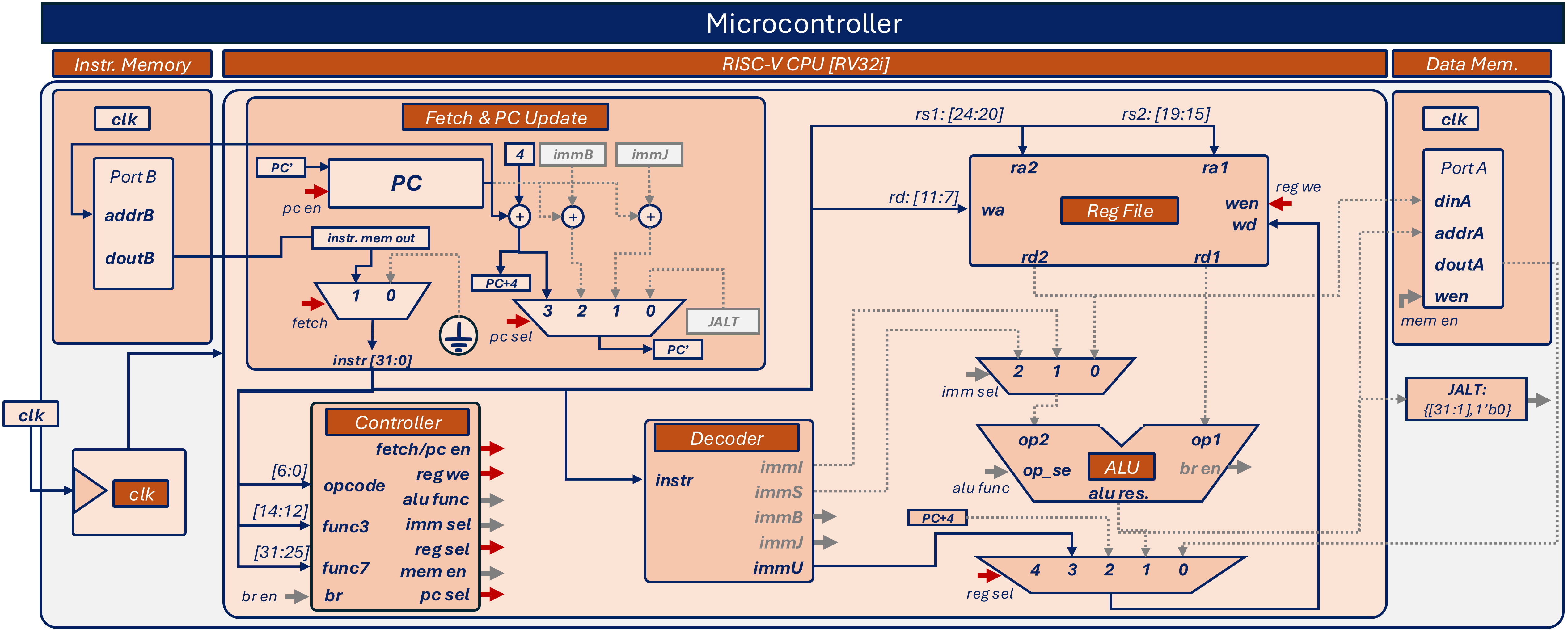
- *The decoder produces a 12-bit signed immJ which is added to PC to give the jump target address*
- *(PC + immJ) is connected to the multiplexor that selects the next PC value, and pc sel is appropriately updated*
- *(PC+4) is connected to the mux that selects the wd input, and reg sel is appropriately updated, writing the return address to rd*

“Jump and Link Register” Operations



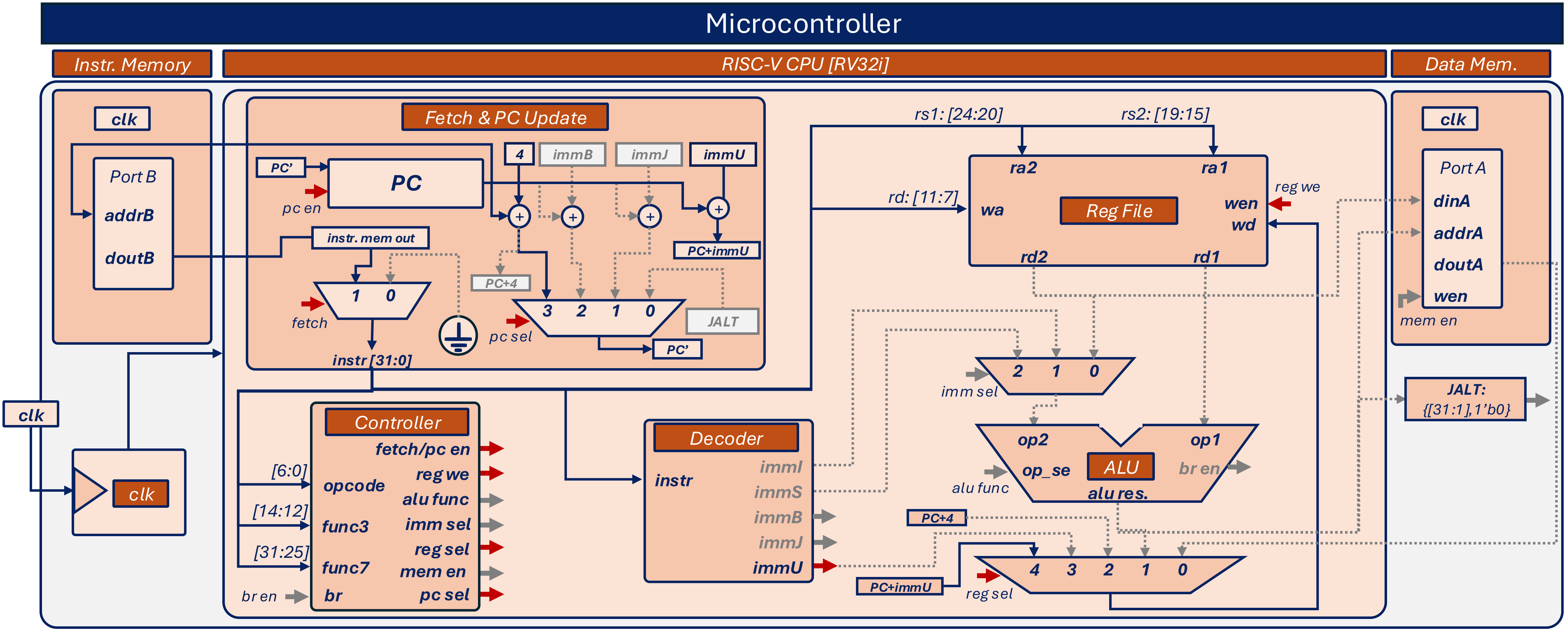
- *The indirect jump instruction JALR (jump and link register) uses the I-type encoding, thus imm sel selects immI as the operand to the ALU*
- *The target address (JALT) is obtained by adding the 12-bit signed I-immediate to rs1, then setting the least-significant bit of the result to zero*
- *(PC + JALT) is connected to the multiplexor that selects the next PC value, and pc sel is appropriately updated*
- *The return address following the jump (PC+4) is written to register rd*

“Load Upper Immediate” Operations



- *LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format*
- *The decoder produces an immU, which is connected to the mux that selects the wd input, and reg sel is appropriately updated*

"Add Upper Immediate to PC" Operations



- (PC + immU) is connected to the multiplexor that selects the next PC value, and pc sel is appropriately updated
- (PC+4) is connected to the mux that selects the wd input, and reg sel is appropriately updated to write this value to the Reg File

Table of Contents

AXI4-Lite Interface

Introduction

- *Advanced eXtensible Interface 4 (AXI4) is a family of buses defined as part of the fourth generation of the ARM Advanced Microcontroler Bus Architectrue (AMBA) standard*
- *AXI4-Lite is a subset of AXI, lacking burst access capability, with a simpler interface than the full AXI4*
- *Xilinx Vivado helps in the creation of custom IP with AXI4 interfaces*

AXI4-Lite Interface Signals

- *The AXI4-Lite interface consists of five channels: Read Address, Read Data, Write Address, Write Data, and Write Response*
- *An AXI4 read transaction using the Read Address and Data channels is shown in figure 1*
- *Similarly, an AXI4 write transaction uses the Write Address, Data, and Response channels*

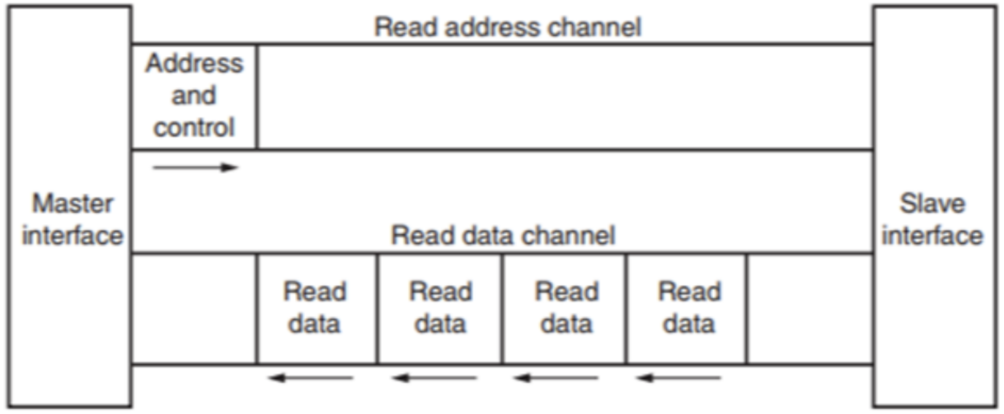


Figure 1. AXI4 Read Transaction.

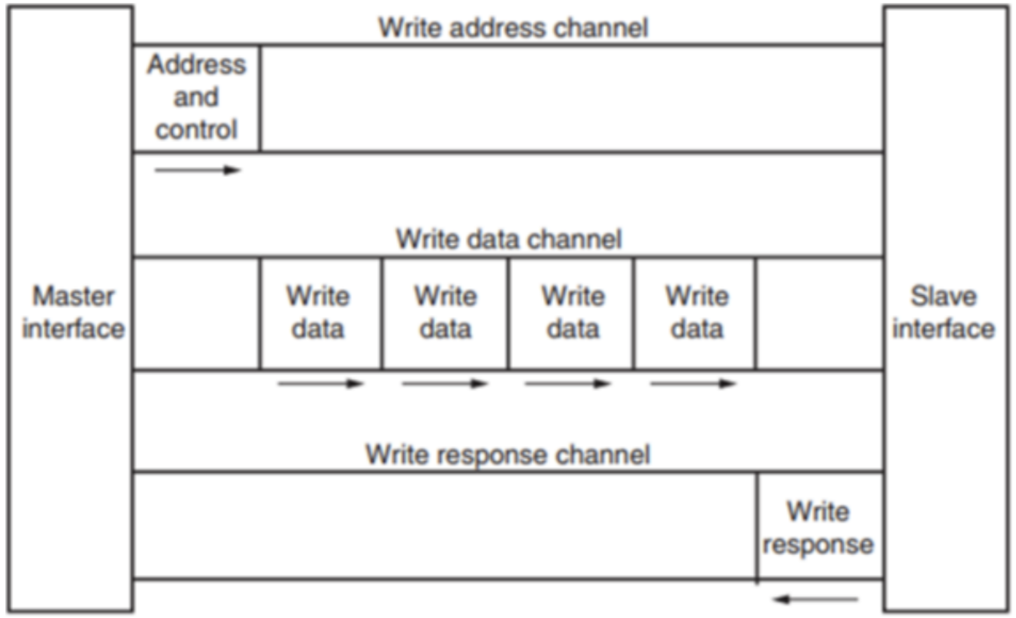


Figure 2. AXI4 Write Transaction.

Sources: *RealDigital: AXI4-Lite Interface (https://www.realdigital.org/doc/a9fee931f7a172423e1ba73f66ca4081)*

AXI4-Lite Read Channel Signals

AXI Read				
Channel	Signal Name		Dir.	Description
Read Address Channel	**Read Addr. Valid**	ARVALID	M → S	• Master generates this signal when Read Address and the control signals are valid
	Read Addr. Ready	ARREADY	M ← S	• Slave generates this signal when it can accept the Read addr. and control signals
	Read Address	ARADDR	M → S	• 32-bit wide usually
	Protection Type	ARPROT	M → S	• Xilinx IP usually ignores as a slave
Read Data Channel	**Read Addr. Valid**	RVALID	M ← S	• Slave generates this signal when Read Data is valid
	Read Addr. Ready	RREADY	M → S	• Master generates this signal when it can accept the Read Data and response
	Read Data	RDATA	M ← S	• 32-bit wide only
	Read Response	RRESP	M ← S	• This signal indicates the status of data transfer

Sources: RealDigital: AXI4-Lite Interface (https://www.realdigital.org/doc/a9fee931f7a172423e1ba73f66ca4081)

AXI4-Lite Write Channel Signals

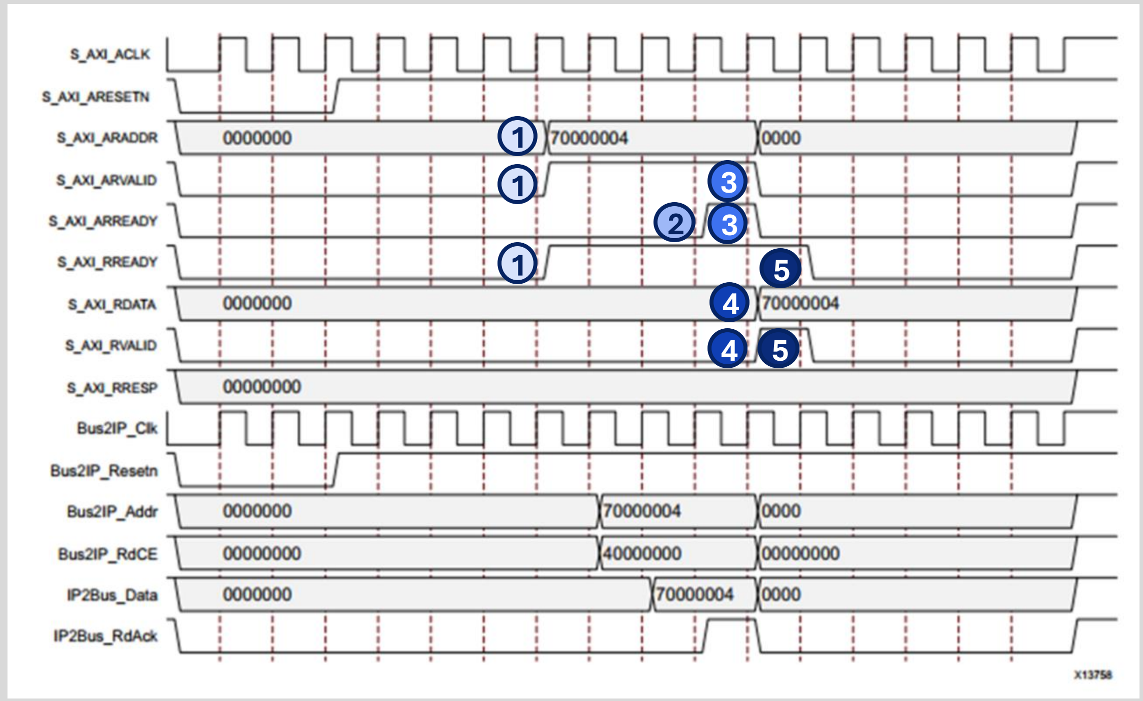
AXI Read				
Channel	Signal Name		Dir.	Description
Write Address Channel	Write Addr. Valid	AWVALID	M → S	• Master generates this signal when Write Address and the control signals are valid
	Write Addr. Ready	AWREADY	M ← S	• Slave generates this signal when it can accept the Write addr. and control signals
	Write Address	AWADDR	M → S	• 32-bit wide usually
	Protection Type	AWPROT	M → S	• Slave IP usually ignores
Write Data Channel	Write Valid	WVALID	M → S	• Master generates this signal when Write Data is valid
	Write Ready	WREADY	M ← S	• Slave generates this signal when it can accept the Write Data and response
	Write Data	WDATA	M ← S	• 32-bit wide only
	Write Strobe	WSTRB	M ← S	• 4-bit signal indicating which of the 4-bytes of Write Data are written
Write Response Channel	Response Valid	BVALID	M ← S	• Slave generates this signal when the write response on the bus is valid
	Response Ready	BREADY	M → S	• Master generates this signal when it can accept a write response
	Write Response	BRESP	M ← S	• Indicates the status of the write transaction

Sources: *RealDigital: AXI4-Lite Interface (https://www.realdigital.org/doc/a9fee931f7a172423e1ba73f66ca4081)*

AXI4-Lite Read Transaction

Below, the sequence for an AXI4-Lite Read is shown and depicted in the figure

1.	*The Master puts an address on the Read Address channel as well as asserting ARVALID, indicating the address is valid, and RREADY, indicating the master is ready to receive data from the slave.*
2.	*The Slave asserts ARREADY, indicating that it is ready to receive the address on the bus.*
3.	*Since both ARVALID and ARREADY are asserted, on the next rising clock edge the handshake occurs, after this the master and slave de-assert ARVALID and the ARREADY, respectively. (At this point, the slave has received the requested address)*
4.	*The Slave puts the requested data on the Read Data channel and asserts RVALID, indicating the data in the channel is valid. The slave can also put a response on RRESP, though this does not occur here*
5.	*Since both RREADY and RVALID are asserted, the next rising clock edge completes the transaction. RREADY and RVALID can now be de-asserted*



AXI4-Lite Write Transaction

Below, the sequence for an AXI4-Lite Write is shown and depicted in the figure

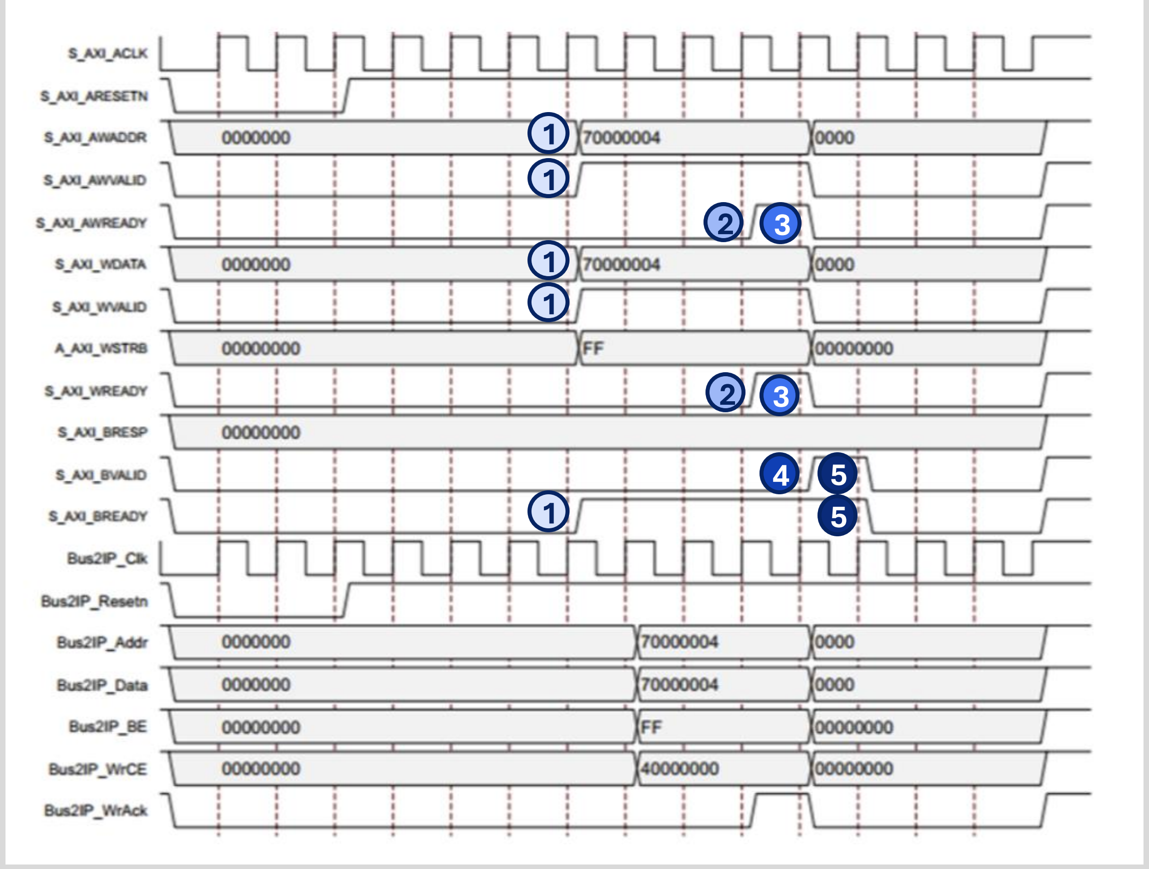
1. *The Master puts an address on the Write Address channel and data on the Write data channel. At the same time, it asserts AWVALID and WVALID indicating the address and data on the respective channels is valid. BREADY is also asserted by the Master, indicating it is ready to receive a response*

2. *The Slave asserts AWREADY and WREADY on the Write Address and Write Data channels, respectively*

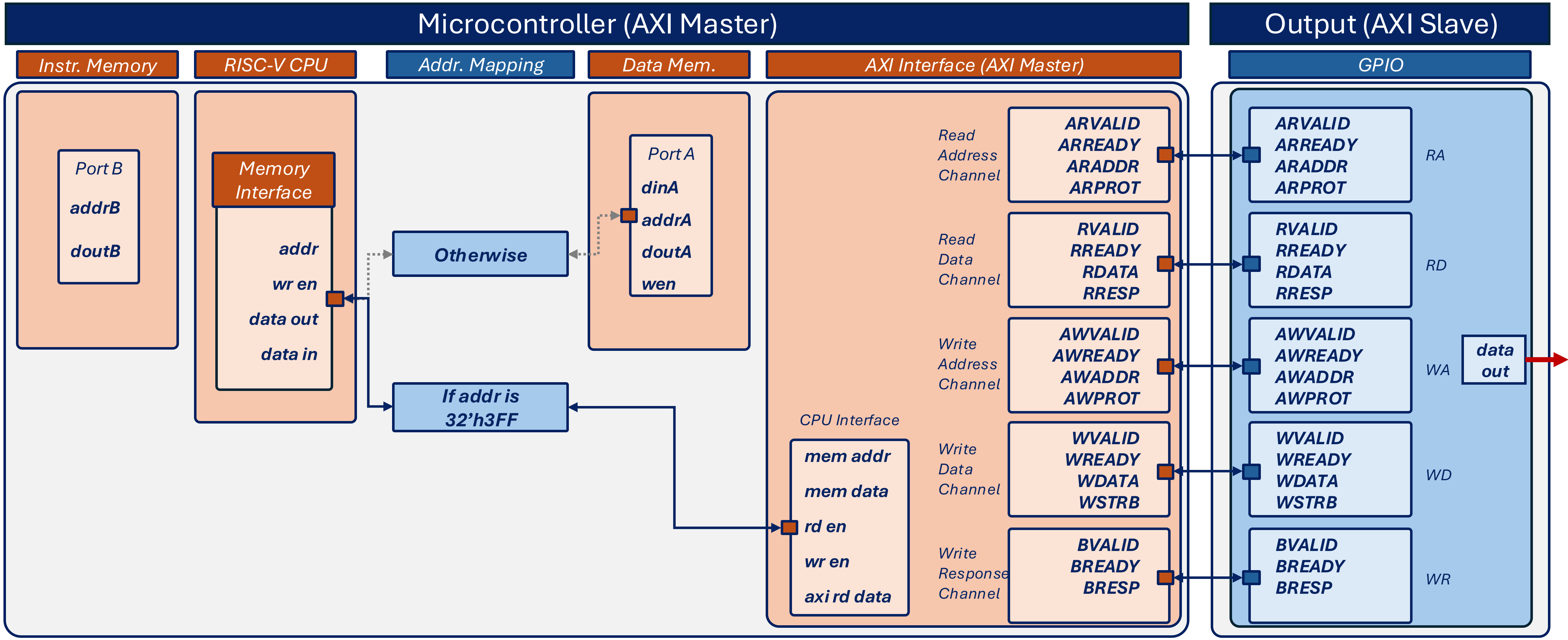
3. *Since Valid and Ready signals are present on both the Write Address and Write Data channels, the handshakes on those channels occur and the associated Valid and Ready signals can be de-asserted. (After both handshakes occur, the slave has the write address and data)*

4. *The Slave asserts BVALID, indicating there is a valid response on the Write response channel. (in this case the response is 2'b00, that being 'OKAY')*

5. *The next rising clock edge completes the transaction, with both the Ready and Valid signals on the write response channel high*



Add AXI4-Lite Master Interface



- In order to interact with external peripheral devices (e.g., GPIO) an AXI master interface is added
- The AXI Master is "address mapped" such that CPU loads/stores are performed via AXI rather than data memory within a certain address range
- E.g., when a store operation is performed, a value is written via AXI to GPIO if the address is 32'h3FF (the final address in our depth of 1024)
- The AXI Master interfaces with the GPIO module, producing relevant signals to perform reads and writes via the AXI handshaking process

Table of Contents

Sample Program

C Code (set_variable.c)

```
#include <stdint.h>

#define DISPLAY_ADDR 0x000003FF

int main() {
     uint32_t *display = (uint32_t *) DISPLAY_ADDR;

     for (int i = 0; i <= 10; ++i) {
          *display = i;
     }
     return 0;
}
```

Program Functionality

- *This program interacts with a memory-mapped GPIO device (hex display) by writing values to the specific memory address 0x000003FF*
- *It iterates through the numbers 0 to 10, writing each number to the display by directly assigning values to the memory-mapped address*

Formatted File (set_variable_formatted.mem)

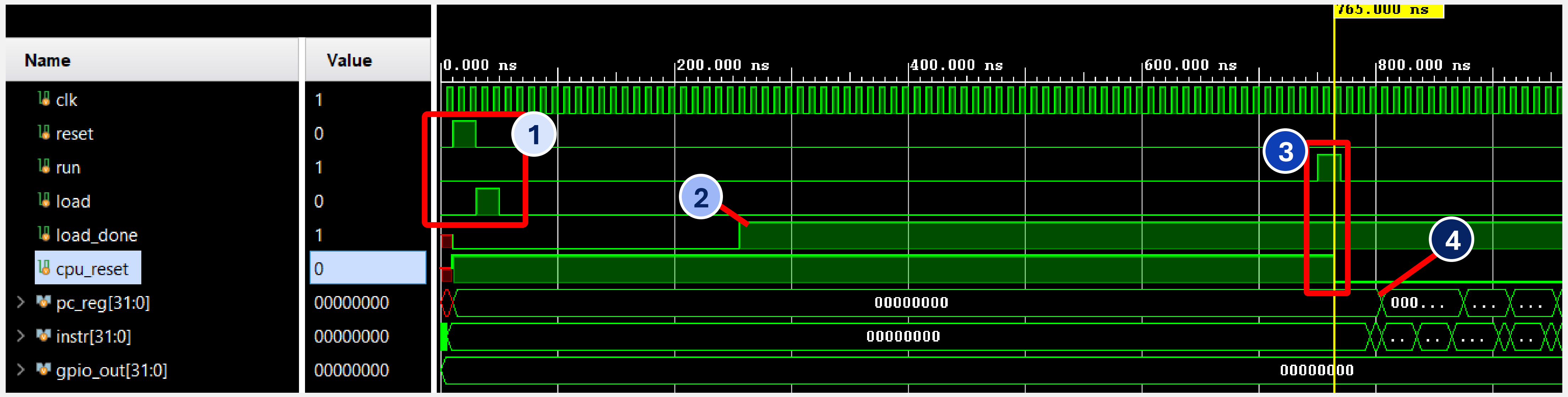
```
// Number of Total Instructions 22 (Hex: 16)

instr_data_reg[0]  = 32'hFE010113; // I-Type
instr_data_reg[1]  = 32'h00812E23; // Store
instr_data_reg[2]  = 32'h02010413; // I-Type
instr_data_reg[3]  = 32'h3FF00793; // I-Type
instr_data_reg[4]  = 32'hFEF42423; // Store
instr_data_reg[5]  = 32'hFE042623; // Store
instr_data_reg[6]  = 32'h01C0006F; // JAL
instr_data_reg[7]  = 32'hFEC42703; // Load
instr_data_reg[8]  = 32'hFE842783; // Load
instr_data_reg[9]  = 32'h00E7A023; // Store
instr_data_reg[10] = 32'hFEC42783; // Load
instr_data_reg[11] = 32'h00178793; // I-Type
instr_data_reg[12] = 32'hFEF42623; // Store
instr_data_reg[13] = 32'hFEC42703; // Load
instr_data_reg[14] = 32'h00A00793; // I-Type
instr_data_reg[15] = 32'hFEE7D0E3; // Branch
instr_data_reg[16] = 32'h00000793; // I-Type
instr_data_reg[17] = 32'h00078513; // I-Type
instr_data_reg[18] = 32'h01C12403; // Load
instr_data_reg[19] = 32'h02010113; // I-Type
instr_data_reg[20] = 32'h00008067; // JALR
instr_data_reg[21] = 32'h00000013; // NOP
```

These bare-metal instructions can now be copied and pasted into the 'Loader' module (instruction_loader.sv), which is responsible for loading the contents of the instr_data_reg register into Instruction Memory

Instruction Loading & Program Initiation

Waveform



1 Reset & Assert Load

- *To begin, "reset" is asserted*
- *Next, "load" is asserted to load the program instructions into instruction memory*

2 Load Done Asserted

- *Once the instruction memory has been loaded, "load_done" is asserted*
- *At this point, "run" can be asserted to initiate program execution*

3 Assert Run

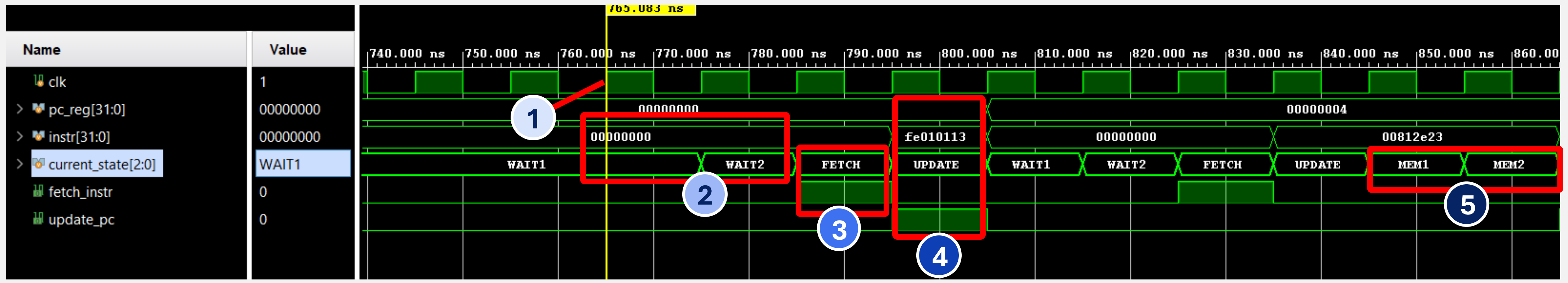
- *The signal "run" is asserted to begin program execution*
- *The result is that "cpu_reset" is de-asserted to allow the cpu to freely fetch and execute instructions*

4 Program Begins Execution

- *After "cpu_reset" is de-asserted, the program begins to be executed*
- *At this point, the PC register ("pc_reg") and corresponding instructions ("instr") begin to update*

State Machine & Instruction Fetch

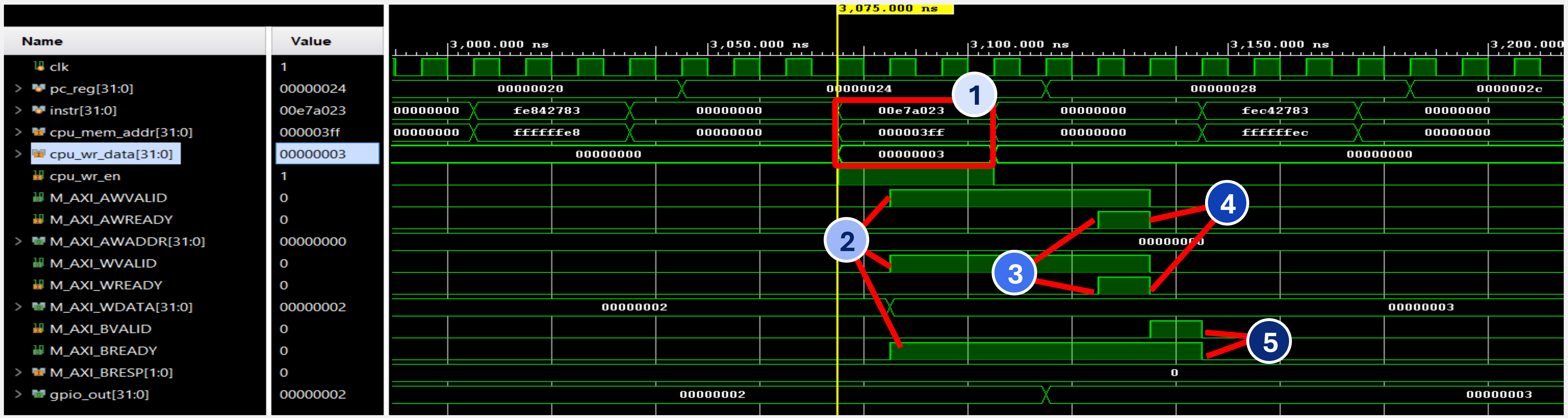
Waveform



1 ***Program Begins Execution***	• *At this rising edge of the clock after "cpu_reset" has been de-asserted, the program execution begins* • *The state-machine begins in the "WAIT1" state, and the state evolves as described below*
2 ***Two Wait States***	• *Two wait states are allowed to account for memory delay to output the instr. corresponding to the PC* • *The next state is "FETCH"*
3 ***Fetch State***	• *A "fetch_instr" signal is asserted to update the instruction corresponding to the current PC* • *The next state is "UPDATE"*
4 ***Update PC***	• *A "update_pc" signal is asserted to update the PC register to the next PC value* • *The state returns to "WAIT1"*
5 ***Memory Wait States***	• *For instructions interfacing with memory (i.e., 'Load' & 'Store'), two additional wait states "Mem 1/2" are entered to account for memory latency*

AXI Write

Waveform



Name	Value
clk	1
pc_reg[31:0]	00000024
instr[31:0]	00e7a023
cpu_mem_addr[31:0]	000003ff
cpu_wr_data[31:0]	00000003
cpu_wr_en	1
M_AXI_AWVALID	0
M_AXI_AWREADY	0
M_AXI_AWADDR[31:0]	00000000
M_AXI_WVALID	0
M_AXI_WREADY	0
M_AXI_WDATA[31:0]	00000002
M_AXI_BVALID	0
M_AXI_BREADY	0
M_AXI_BRESP[1:0]	0
gpio_out[31:0]	00000002

1 Store Instruction

- *The current instruction is a "Store" operation, where the value 32'h3 is written to address 32'h3FF*
- *The GPIO module is memory-mapped to address 32'h3FF, so an AXI-write to GPIO is initiated*

2 Valid Signals

- *The Master puts an address on the Write Address channel and data on the Write data channel*
- *At the same time, it asserts AWVALID, WVALID, and BREADY*

3 Ready Signals

- *The Slave asserts AWREADY and WREADY on the Write Address and Write Data channels*

4 Handshake

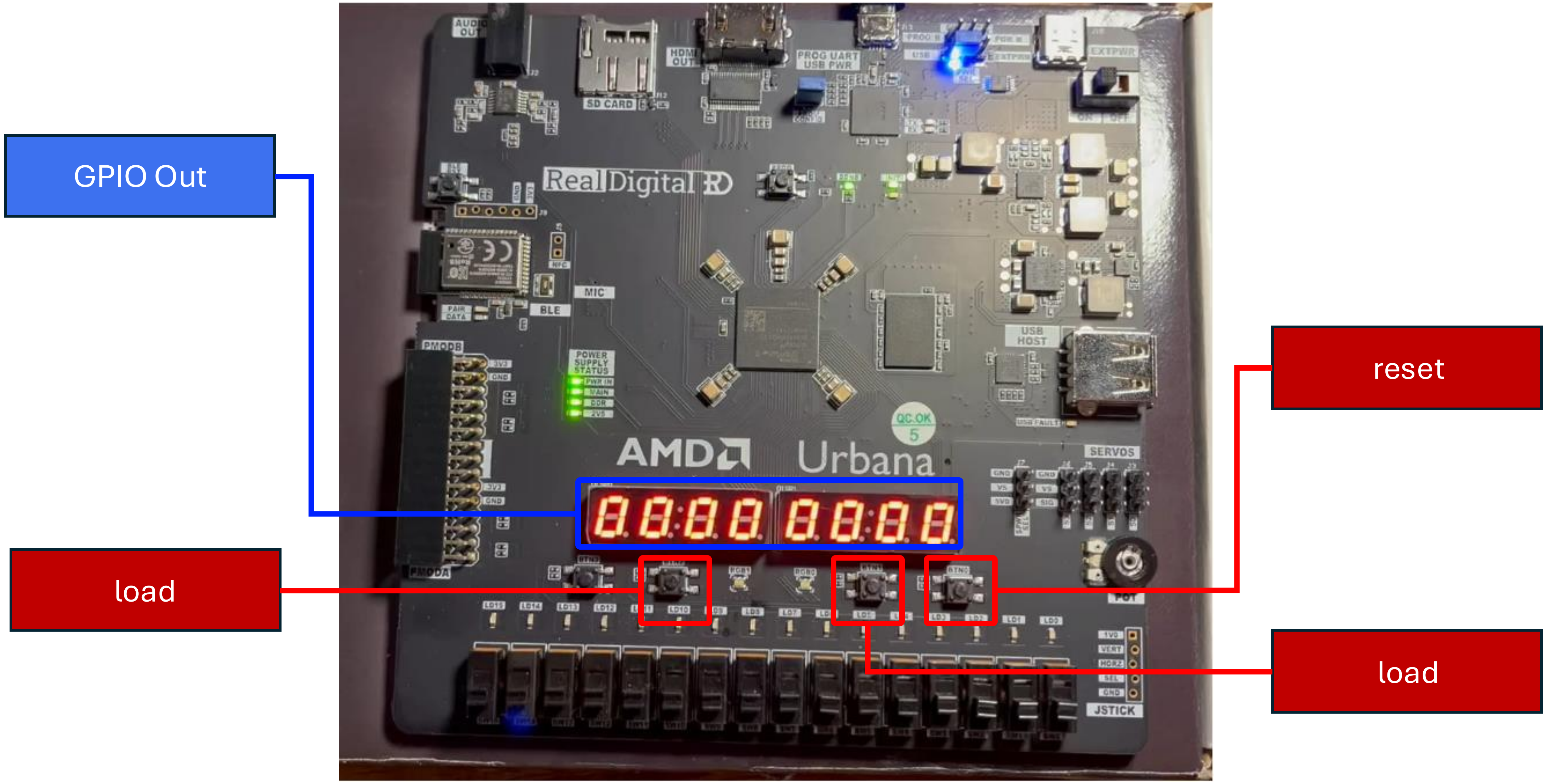
- *The handshakes on those channels occur and the associated Valid and Ready signals are de-asserted*
- *After both handshakes occur, the slave has the write address and data*

5 Response Valid

- *The Slave asserts BVALID, indicating there is a valid response on the Write response channel*
- *The next rising clock edge completes the transaction*

Table of Contents

Input / Output



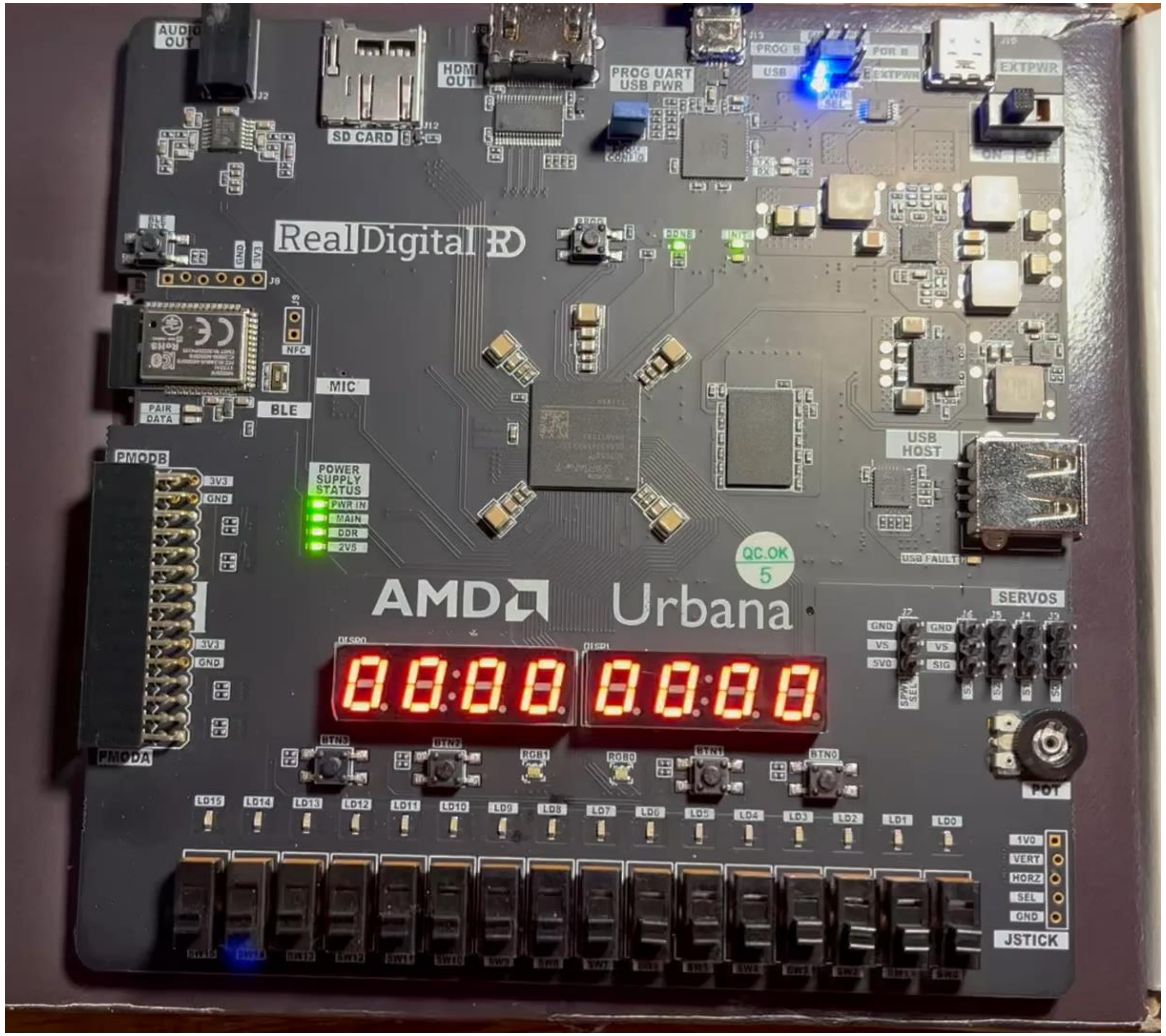

RealDigital
AMD
Urbana
POWER SUPPLY STATUS
SD CARD
HDMI OUT
PROG UART USB PWR
EXTPWR
USB HOST
SERVOS
PMODB
PMODA
MIC
BLE
POT
JSTICK
QC.OK 5

Table of Contents

Download Cygwin

What is Cygwin?

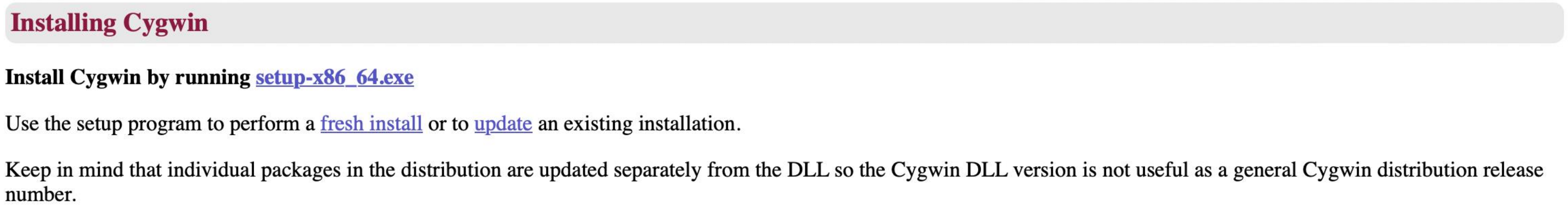
- *Cygwin is a collection of open-source tools that provide a Linux-like environment on Windows, allowing Linux applications to be compiled and run on Windows*
- *It provides a command-line interface similar to a Unix shell, making it easier to use Linux-based tools and scripts on a Windows system*

Why do we need Cygwin?

- *The RISC-V toolchain is designed to be built in a Unix-like environment; thus Cygwin is necessary as Windows does not natively support the same development tools as Unix-based systems*
- *The RISC-V toolchain depends on a number of libraries and tools that are commonly available in Linux environments, such as GNU Compiler Collection (GCC), Make, and various others that will be discussed later*

How to Download Cygwin

- *Visit the Cygwin Website (https://www.cygwin.com/) and click on the link to download (setup-x86_64.exe)*



Installing Cygwin

Install Cygwin by running setup-x86_64.exe

Use the setup program to perform a fresh install or to update an existing installation.

Keep in mind that individual packages in the distribution are updated separately from the DLL so the Cygwin DLL version is not useful as a general Cygwin distribution release number.

Run Cygwin Installer & Select Packages

Initial Setup

- *Once downloaded, open the Cygwin installer*
- *The installer will ask to 'Choose a Download Source', 'Select Root Install Directory', etc.*
- *It is suitable to click "Next" through each page using the default options*

Package Selection

- *In 'Select Packages' page, select the most recent version of the following packages for the RISC-V toolchain build*

Package	Description
automake	Wrapper for multiple versions of Automake
binutils	GNU assembler, linker, and similar utilities
ca-certificates	CA root certificates
gcc-core	GNU Compiler Collection (C, OpenMP)
gcc-g++	GNU Compiler Collection (C++)
libatomic1	GCC C11/C++ 11 locked atomics runtime library
libcharset1	Unicode iconv() implementation
libcurl4	Multi-protocol file transfer library (runtime)
libdeflate0	A library for fast, whole-buffer DEFLATE-based compression and (runtime)
libgcc1	GCC C runtime library
libgcrypt20	GnuPG cryptography library
libgomp1	GCC OpenMP runtime library
libgpg-error0	GnuPG error code library
libiconv-devel	Unicode iconv() implementation
libjpeg8	JPEG library with SIMD acceleration (runtime)

Package	Description
libpkgconf5	Pkgconf library iplementation
libquadmath0	GCC Quad-Precision Math runtime library
libstdc++6	GCC C++ runtime library
mintty	Terminal emulator with native Windows look and feel
openssh	The OpenSSH server and client programs
pkg-config	Package compiler and linker metadata tool (alternative to pkg-config)
pkgconf	Package compiler and linker metadata tool (alternative to pkg-config)

- *Once you have selected these packages, click "Next" to proceed, at which time the installer will download and install all selected packages*

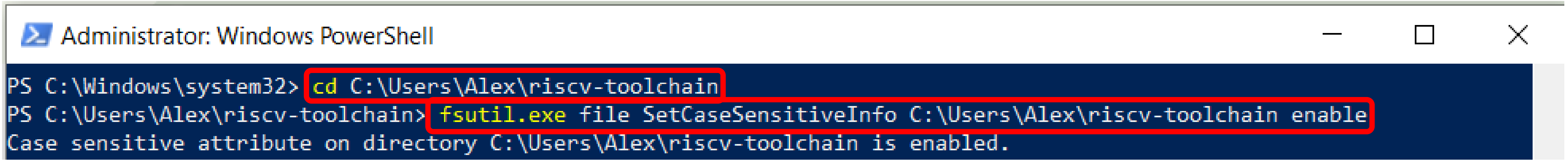
Create Case-Sensitive File Directory

Understanding Case-Sensitivity in Windows

- *In Linux, the filesystem is case-sensitive, meaning 'File.txt' and 'file.txt' are considered distinct*
- *Windows by default, however, is not case-sensitive, so these two files would be treated as the same*
- *The installation of the RISC-V toolchain relies on case-sensitivity, and an attempt to build it in a non-case-sensitive directory would likely lead to errors*

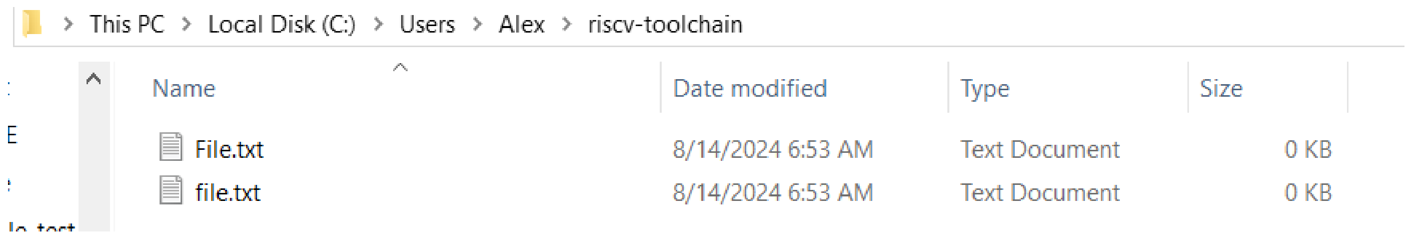
How to Enable Case-Sensitivity

- *Create a folder in which you intend to install the RISC-V toolchain, for example, "C:\Users\Alex\riscv-toolchain"*
- *In your Windows Desktop, click 'Start', search for 'Windows PowerShell', right click, and select 'Run as Administrator'*
- *Run the following two commands to navigate to the path of the previously mentioned folder and enable case-sensitivity*



```
PS C:\Windows\system32> cd C:\Users\Alex\riscv-toolchain
PS C:\Users\Alex\riscv-toolchain> fsutil.exe file SetCaseSensitiveInfo C:\Users\Alex\riscv-toolchain enable
Case sensitive attribute on directory C:\Users\Alex\riscv-toolchain is enabled.
```

- *Restart the system, then for confirmation create the following two files in the case-sensitive folder: 'File.txt' and 'file.txt'*
- *If the folder has been properly made case-sensitive, you should see two distinct files in the folder*



Name	Date modified	Type	Size
File.txt	8/14/2024 6:53 AM	Text Document	0 KB
file.txt	8/14/2024 6:53 AM	Text Document	0 KB

Clone & Build the RISC-V Toolchain Repository

Toolchain setup

- *Navigate to the directory you made case-sensitive, and clone the RISC-V GNU toolchain repository from GitHub*

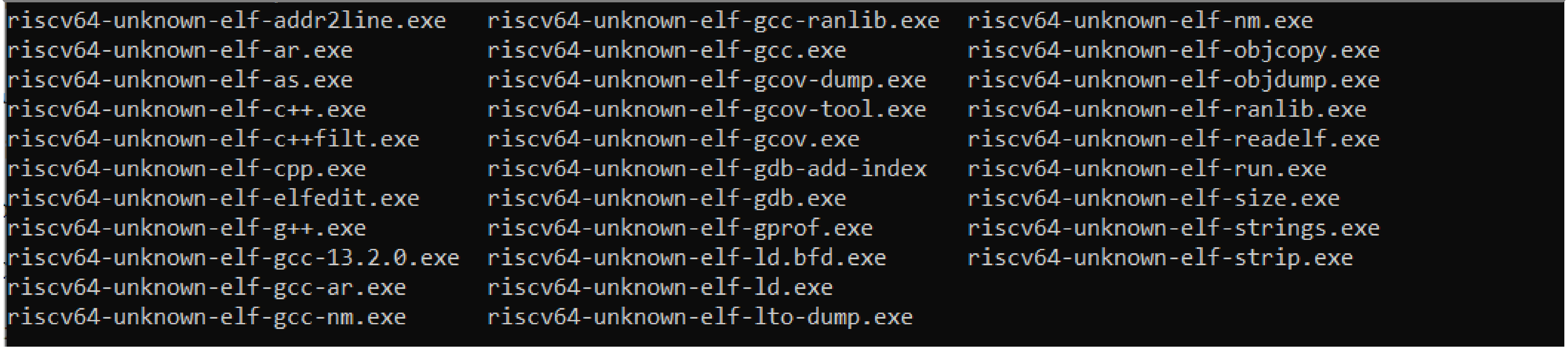
```
cd C:\Users\Alex\riscv-toolchain
git clone https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain
```

- *Configure the build to install the toolchain in a specific directory (e.g., '/opt/riscv'), and start the build process*

```
./configure –prefix=/opt/riscv
make
```

- *Upon completion, navigate to the following folder and confirm the existence of the following files: "riscv64-unknown-elf-gcc.exe", "riscv64-unknown-elf-as.exe", "riscv64-unknown-elf-objcopy.exe"*

```
C:\Users\Alex>cd C:\Users\Alex\riskv-toolchain\riscv\toolchain\bin
C:\Users\Alex\riskv-toolchain\riscv\toolchain\bin>ls
```



```
riscv64-unknown-elf-addr2line.exe   riscv64-unknown-elf-gcc-ranlib.exe  riscv64-unknown-elf-nm.exe
riscv64-unknown-elf-ar.exe          riscv64-unknown-elf-gcc.exe         riscv64-unknown-elf-objcopy.exe
riscv64-unknown-elf-as.exe          riscv64-unknown-elf-gcov-dump.exe   riscv64-unknown-elf-objdump.exe
riscv64-unknown-elf-c++.exe         riscv64-unknown-elf-gcov-tool.exe   riscv64-unknown-elf-ranlib.exe
riscv64-unknown-elf-c++filt.exe     riscv64-unknown-elf-gcov.exe        riscv64-unknown-elf-readelf.exe
riscv64-unknown-elf-cpp.exe         riscv64-unknown-elf-gdb-add-index   riscv64-unknown-elf-run.exe
riscv64-unknown-elf-elfedit.exe     riscv64-unknown-elf-gdb.exe         riscv64-unknown-elf-size.exe
riscv64-unknown-elf-g++.exe         riscv64-unknown-elf-gprof.exe       riscv64-unknown-elf-strings.exe
riscv64-unknown-elf-gcc-13.2.0.exe  riscv64-unknown-elf-ld.bfd.exe      riscv64-unknown-elf-strip.exe
riscv64-unknown-elf-gcc-ar.exe      riscv64-unknown-elf-ld.exe
riscv64-unknown-elf-gcc-nm.exe      riscv64-unknown-elf-lto-dump.exe
```

Files Used for RISC-V Instructions

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Relevant Files

File *[source]*	***Description***
riscv64-unknown-elf-gcc.exe *[RISC-V toolchain]*	• This is a compiler that converts C source code (.c file) into assembly code (.s file)
riscv64-unknown-elf-as.exe *[RISC-V toolchain]*	• This is an assembler that converts assembly code (.s file) into an object file (.o file) • Translates the assembly instructions into machine code, which is stored in the object file
riscv64-unknown-elf-objcopy.exe *[RISC-V toolchain]*	• This is a binary utility that manipulates and converts object files • In this case, objcopy is used to convert the object file (.o) into a memory initialization file (.mem)
format_instructions.py *[Custom]*	• This Python script converts the MEM file (file.mem), such that each instruction occupies the contents of a 32-bit register, which is later loaded into the corresponding index of Instr. Memory • The output (file_formatted.mem) contents are copied & pasted into the 'Instruction Loader' .sv file

Original File (file.mem)

```
@00000000
13 01 01 FE 23 2E 81 00 13 04 01 02 93 07 F0 3F
23 24 F4 FE
```

Formatted File (file_formatted.mem)

```
// Number of Instructions: 5 (Hex: 05)
instr_data_reg[0] = 32'hFE010113 // I-Type
instr_data_reg[1] = 32'h00812E23 // Store
instr_data_reg[2] = 32'h02010413 // I-Type
instr_data_reg[3] = 32'h3FF00793 // I-Type
instr_data_reg[4] = 32'hFEF42423 // Store
```

Converting C Code to RV32i Bare-Metal Instructions

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

1 Create Your Project Folder

- Begin by creating a folder where you will save your C file and other related scripts

```
C:\Users\Alex>mkdir riscv_code
```

2 Import the Python Script

- Copy the 'format_instructions.py' script into the newly created folder
- This script will be used later to format the .mem file output

3 Create & Save Your C File

- Navigate to the newly created folder, create a C file, and populate it with your c code using an IDE and save

```
C:\Users\Alex>cd C:\Users\Alex\riscv_code
C:\Users\Alex\riscv_code>type nul > file.c
```

4 Compile the C File

- Execute the following command to compile the C file to Assembly

```
C:\Users\Alex\riscv_code>"C:\Users\Alex\riskv-toolchain\riscv\toolchain\bin\riscv64-unknown-elf-gcc.exe" -march=rv32i -
mabi=ilp32 -S -o file.s file.c
```

5 Compile the C File

- Execute the following command to compile the C file to Assembly

```
C:\Users\Alex\riscv_code>"C:\Users\Alex\riskv-toolchain\riscv\toolchain\bin\riscv64-unknown-elf-as.exe" -march=rv32i -
mabi=ilp32 -o file.o file.s
```

6 Convert to a Memory File

- Execute the following command to convert the Assembly to a Verilog .mem file

```
"C:\Users\Alex\riskv-toolchain\riscv\toolchain\bin\riscv64-unknown-elf-objcopy.exe" -O verilog file.o file.mem
```

7 Format the Memory File

- Execute the following command to format file.mem using the previously added Python script

```
C:\Users\Alex>python "C:\Users\Alex\riscv_code\format_instructions.py" file.mem file_formatted.mem
```

Important: *Before running the commands, double-check that the directory names and paths to the RISC-V toolchain and your project folder are correct*

Formatted Instructions Output

C Code (set_variable.c)

```
#include <stdint.h>

#define DISPLAY_ADDR 0x000003FF

int main() {
     uint32_t *display = (uint32_t *) DISPLAY_ADDR;

     for (int i = 0; i <= 10; ++i) {
          *display = i;
     }
     return 0;
}
```

Program Functionality

- *This program interacts with a memory-mapped GPIO device (hex display) by writing values to the specific memory address 0x000003FF*
- *It iterates through the numbers 0 to 10, writing each number to the display by directly assigning values to the memory-mapped address*

Formatted File (set_variable_formatted.mem)

```
// Number of Total Instructions 22 (Hex: 16)

instr_data_reg[0]  = 32'hFE010113; // I-Type
instr_data_reg[1]  = 32'h00812E23; // Store
instr_data_reg[2]  = 32'h02010413; // I-Type
instr_data_reg[3]  = 32'h3FF00793; // I-Type
instr_data_reg[4]  = 32'hFEF42423; // Store
instr_data_reg[5]  = 32'hFE042623; // Store
instr_data_reg[6]  = 32'h01C0006F; // JAL
instr_data_reg[7]  = 32'hFEC42703; // Load
instr_data_reg[8]  = 32'hFE842783; // Load
instr_data_reg[9]  = 32'h00E7A023; // Store
instr_data_reg[10] = 32'hFEC42783; // Load
instr_data_reg[11] = 32'h00178793; // I-Type
instr_data_reg[12] = 32'hFEF42623; // Store
instr_data_reg[13] = 32'hFEC42703; // Load
instr_data_reg[14] = 32'h00A00793; // I-Type
instr_data_reg[15] = 32'hFEE7D0E3; // Branch
instr_data_reg[16] = 32'h00000793; // I-Type
instr_data_reg[17] = 32'h00078513; // I-Type
instr_data_reg[18] = 32'h01C12403; // Load
instr_data_reg[19] = 32'h02010113; // I-Type
instr_data_reg[20] = 32'h00008067; // JALR
instr_data_reg[21] = 32'h00000013; // NOP
```

These bare-metal instructions can now be copied and pasted into the 'Loader' module (instruction_loader.sv), which is responsible for loading the contents of the instr_data_reg register into Instruction Memory