KING'S College LONDON

Hardware Design Lab

Task 5: Branching and IO

Instructor: Dr. Haiyu Mao
TA: Zihao Pu
Ali Alsarraf
Stephen Johannesson

Mar 5, 2026

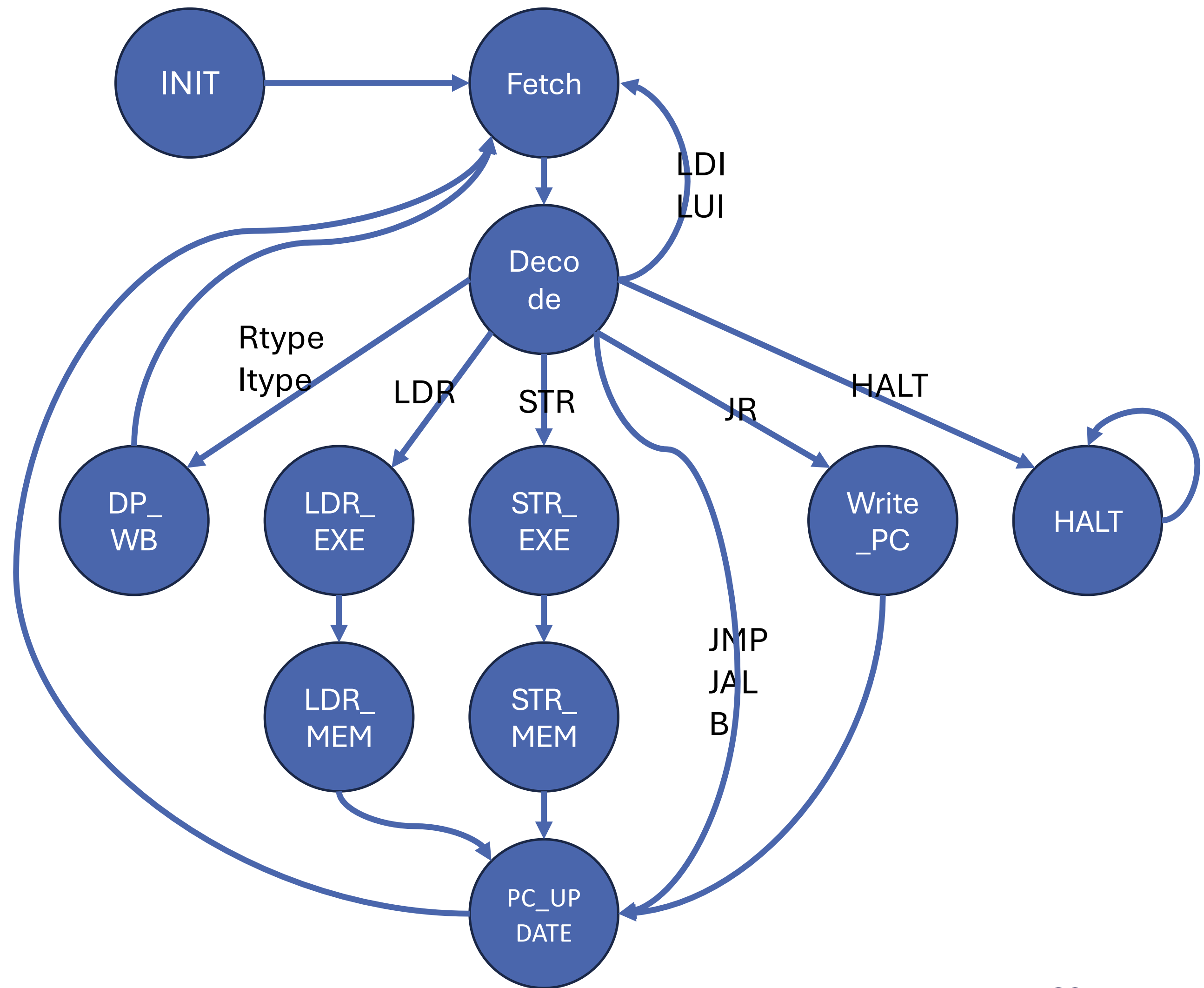
Gratefully acknowledge:
Prof. Onur Mutlu (ETH)
Dr. Yair Linn (TRIUMF Canada)
Prof. Tor Aamodt (UBC)

FSM

TO SUPPORT MEMORY ACCESS AND BRANCHING

Example FSM

You can implement FSM in your own way. Especially if you choose to use a different datapath design.



instruction[7:5] (Ra)
instruction[2:0] (Rb)
instruction[10:8] (Rd)
rb_sel
Instruction Register
instruction[7:0]
tailext8
signext8
txmm8
sximm8
pc_plus1
datapath_in
vsel
data_in
rf_wen
clk
rst
waddr raddrb raddra
wdata
wen
RegFile
8x16 bit
R0-R7
rdatab
rdataa
out_b
out_a
shift_op
sximm5
bsel
ALUOp
shifter
b_shifted
B_oprand
A_oprand
16'b0
asel
ALU
alu_out
load_a
load_y
load_s
status
status_out
datapath_out
dp_gate
data_bus
Datapath
load_ir
data_bus
mem_gate
mem_ena
PC+1
PC+1+SXIMM8
ALU_Out
16'b0
pc_sel
PC_next
load_pc
clk
PC
addr_sel
6'h00
mem_addr [15:10]
mem_wen
mem_ena
mem_addr[9:0]
DI
WEA
ENA
ADDR
clk
DQ
MEM

IO
load_ir
ADD R0, R1, R2
Cycle 1: Fetch
0
1
MEM[0]
PC INCREMENT

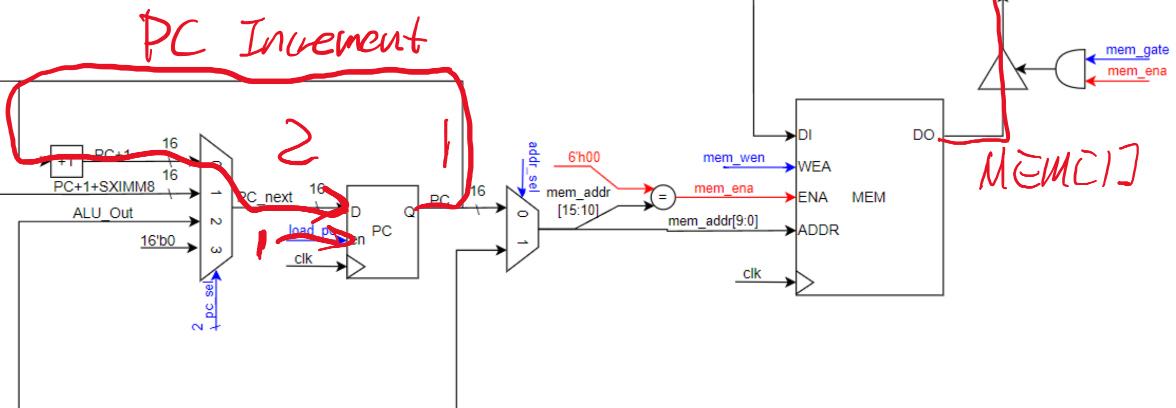
Demo

Cycle 2 decode

Demo

Cycle3: DP-WB

Demo



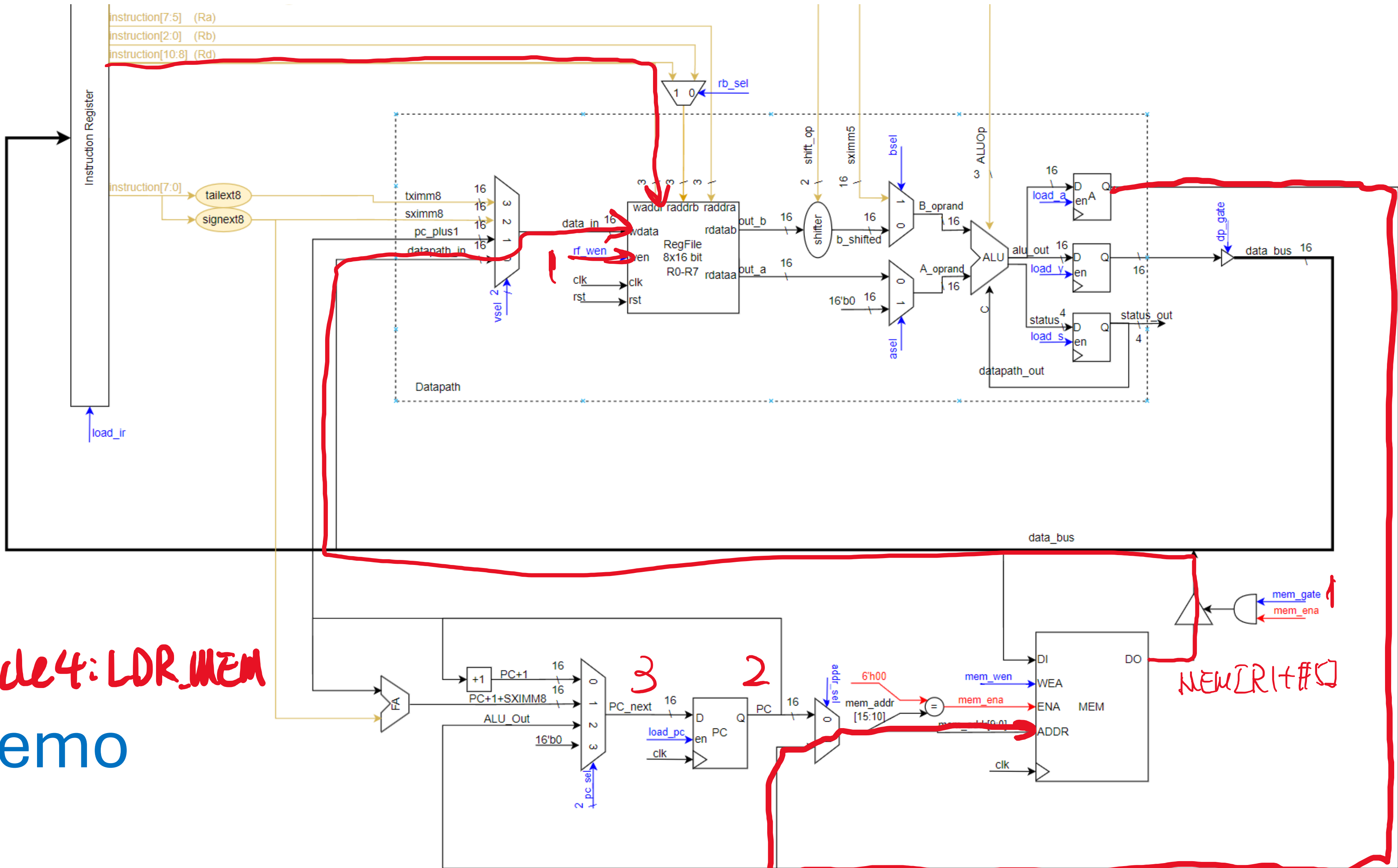
Demo

Cycle 2: Decode

Demo

Cycle 3: LDR_EXE

Demo



Demo

Cycles: PC_update

3 2

MEM[R1+#5]

Demo

Demo

STR Rd*,[Ra,5]

Cycle1: Fetch

Instruction Register

load_ir 1

instruction[7:5] (Ra)
instruction[2:0] (Rb)
instruction[10:8] (Rd)
instruction[7:0]

tailext8
signext8

rb_sel

txtimm8
sximm8
pc_plus1
datapath_in
vsel

data_in
rf_wen
clk
rst

waddr raddrb raddra
wdata
wen
clk
rst
RegFile
8x16 bit
R0-R7
rdatab
rdataa

out_b
out_a
shift_op
shifter
sximm5
b_shifted
bsel
B_oprand
A_oprand
16'b0
asel
ALUOp
ALU
alu_out
load_a
load_y
load_s
status
status_out
datapath_out
dp_gate
data_bus
Datapath

data_bus
mem_gate
mem_ena
1
DI
DO
WEA
ENA
ADDR
MEM
clk
mem_wen
mem_ena
mem_addr[9:0]
6'h00
mem_addr
[15:10]
addr_sel
PC
load_pc
clk
PC+1
+1
PC+1+SXIMM8
ALU_Out
16'b0
PC_next
pc_sel
FA

MEM[2]

1 2 3

Cycle 2: decode

A = Ra+5

MEM[3]

Demo

Cycle3: STR-EXE

load Rd* to Y

Addr=Ra+5

MEM [3]

Demo

Cycle 4: STR MEM

Demo

Cycles: PC_UPDATE

4

3

MEM[Addr]

Demo

JMP #10

Demo

Cycle 1: Fetch

PC INCREMENT

1

3

4

MEM[3]

JMP #10

MEM [3]

Demo

Cycle 2: Decode

Instruction Register
instruction[7:5] (Ra)
instruction[2:0] (Rb)
instruction[10:8] (Rd)
instruction[7:0]
tailext8
signext8
load_ir
rb_sel
txImm8
sxImm8
pc_plus1
datapath_in
vsel
data_in
rf_wen
clk
rst
waddr raddrb raddra
wdata
wen
RegFile
8x16 bit
R0-R7
rdatab
rdataa
out_b
out_a
shift_op
sximm5
bsel
shifter
b_shifted
B_oprand
A_oprand
16'b0
asel
ALUOp
ALU
alu_out
load_a
load_y
load_s
status
status_out
datapath_out
dp_gate
data_bus
Datapath
mem_gate
mem_ena
+1
PC+1
PC+1+SXIMM8
ALU_Out
PC_next
pc_sel
load_pc
PC
addr_sel
mem_addr
[15:10]
6'h00
mem_wen
mem_addr[9:0]
DI
DO
WEA
ENA
ADDR
MEM
clk

JMP #10

Demo

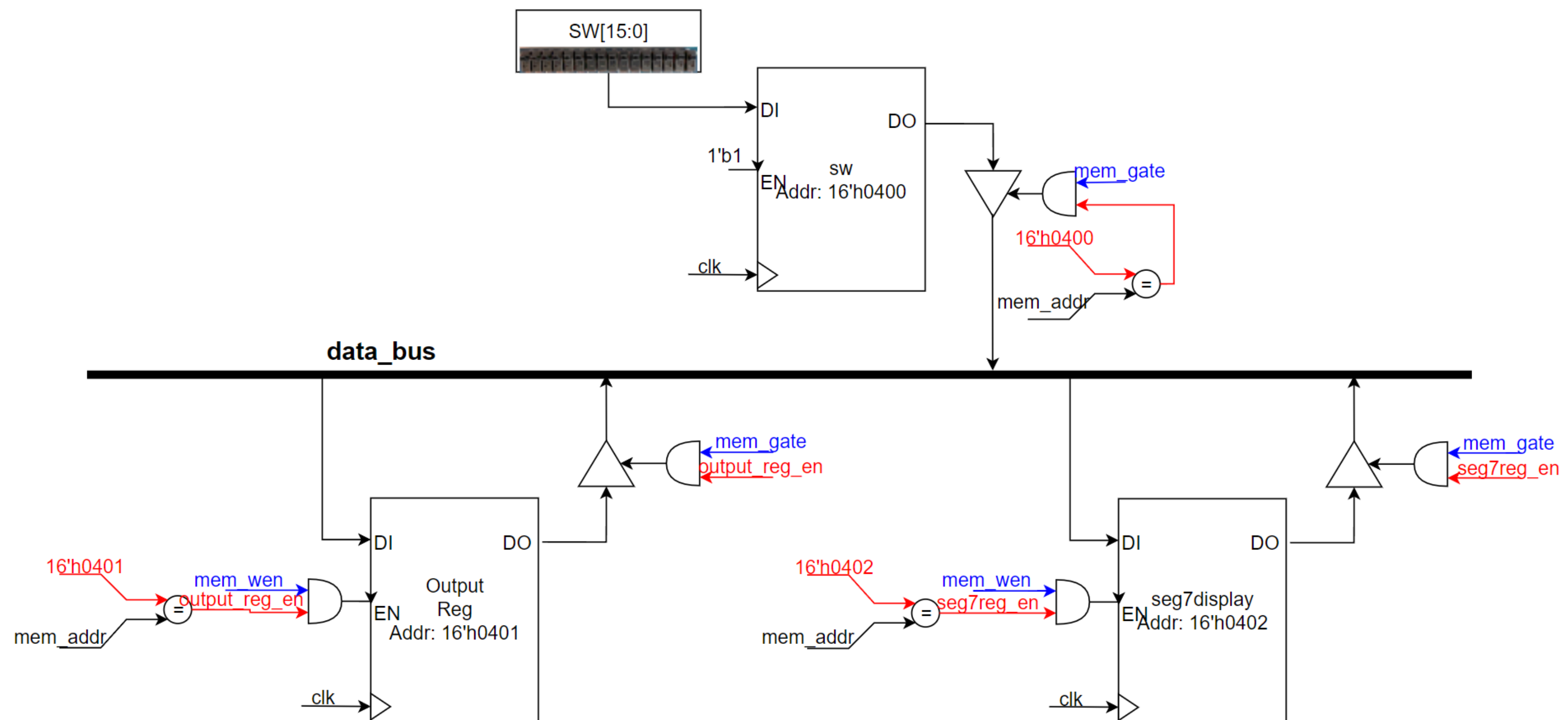
Cycle3: PC_UPDATE

MEM[4]

IO

MEMORY-MAPPED PERIPHERALS

Memory-Mapped Peripherals: Special Function Registers



Memory Map for RISKing16

- Memory-mapping is a typical method to design peripherals. By giving the peripheral a memory address, the processor can deal with IO simply by reading and writing memory addresses like main memory.
- Example usage:

Address Rage	Description	RW
0xFFFF 0x0800	Unused Space	NA
0x07FF 0x0403	Reserved SFR space	NA
0x0402	7-segment display register	RW
0x0401	Output Register (LED)	RW
0x0400	Input Register	Read Only
0x03FF 0x0000	Main Memory	RW

```
; Load SFR base address (0x04 << 8 = 0x0400) into R0
LUI R0, 0x04            ; R0 = 0x0400 (SFR base address)

; Read input from switches
LDR R1, [R0, 0]         ; R1 = INPUT_REG (offset 0 from base)

; Write to LEDs
STR R2, [R0, 1]         ; OUTPUT_REG = R2 (offset 1 from base)

; Write to 7-segment display
STR R3, [R0, 2]         ; SEG7_REG = R3 (offset 2 from base)
```

R-type Instructions (Register Operations)

- example

```
ADD R1, R2, R3          ; R1 = R2 + R3 (no shift)
SUB R4, R5, R6          ; R4 = R5 - R6
ADD R0, R1, LSL(R2)     ; R0 = R1 + (R2 << 1)
ADD R0, R1, R2, LSL     ; R0 = R1 + (R2 << 1)  (alternative syntax)
AND R3, R4, ASR(R5)     ; R3 = R4 & (R5 >>> 1)
```

I-type Instructions (Register Operations)

- example

```
ADDI R1, R2, 10         ; R1 = R2 + 10
SUBI R3, R4, 5          ; R3 = R4 - 5
ADDI R5, R5, -1         ; R5 = R5 - 1 (decrement)
```

Load Immediate Instructions

- example

```
LDI R1, 100              ; R1 = 0x0064 (100, positive – bit 7 is 0)
LDI R2, 0xFF             ; R2 = 0xFFFF (-1, sign-extended – bit 7 is 1)
LUI R3, 0x04             ; R3 = 0x0400 (0x04 << 8)
LUI R4, 0x12             ; R4 = 0x1200 (0x12 << 8)
```

Memory Access Instructions

- example

```
LDR R3, [R1, 0]         ; R3 = Mem[R1 + 0]
LDR R4, R1, 5           ; R4 = Mem[R1 + 5] (alternative syntax)
STR R5, [R1, -2]        ; Mem[R1 - 2] = R5
```

Jump and Branch Instructions

- example

```
JMP loop_start          ; Unconditional jump
JAL subroutine          ; Call subroutine (return address in R7)
JR                      ; Return from subroutine (PC = R7)
BEQ skip                ; Branch if equal (Z flag set)
BNE continue            ; Branch if not equal (Z flag clear)
```

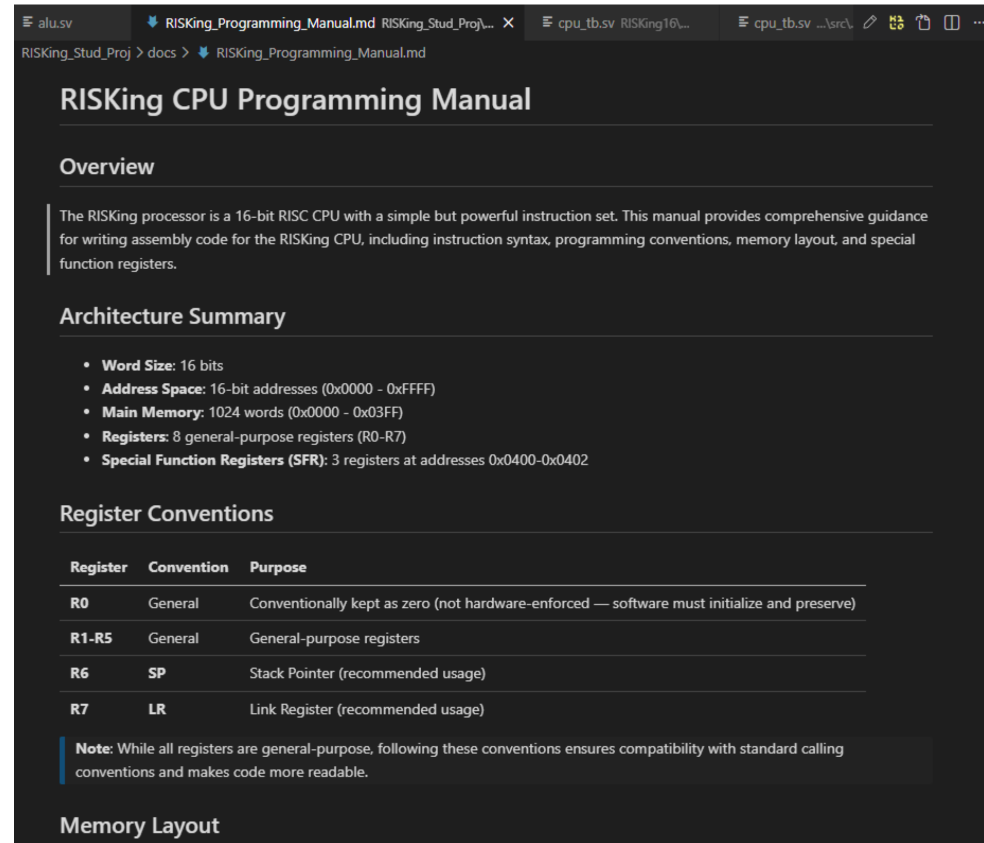
Special Instructions

- example

```
HALT                    ; Stop CPU execution
NOP                     ; No operation
MOV R1, R2              ; R1 = R2 (assembled as ADDI R1, R2, 0)
```

For complete instruction set:

RISC Processor Instruction Set Reference.xlsx
RISKing_Programming_Manual.md



RISKing CPU Programming Manual

Overview

> The RISKing processor is a 16-bit RISC CPU with a simple but powerful instruction set. This manual provides comprehensive guidance for writing assembly code for the RISKing CPU, including instruction syntax, programming conventions, memory layout, and special function registers.

Architecture Summary

- **Word Size**: 16 bits
- **Address Space**: 16-bit addresses (0x0000 - 0xFFFF)
- **Main Memory**: 1024 words (0x0000 - 0x03FF)
- **Registers**: 8 general-purpose registers (R0-R7)
- **Special Function Registers (SFR)**: 3 registers at addresses 0x0400-0x0402

Register Conventions

Register	Convention	Purpose
R0	General	Conventionally kept as zero (not hardware-enforced — software must initialize and preserve)
R1-R5	General	General-purpose registers
R6	**SP**	Stack Pointer (recommended usage)
R7	**LR**	Link Register (recommended usage)

> **Note**: While all registers are general-purpose, following these conventions ensures compatibility with standard calling conventions and makes code more readable.

Memory Layout

Assembly Program with RISKing: Syntax

Basic Structure
[label:] instruction [operands]   ; comment
.DIRECTIVE [arguments]

Labels
End with colon (:)
Used for jump targets and data references
Case-sensitive

Comments
Single-line comments: ; or //
Comments are ignored by the assembler

Assembler Directives
.ORG address: Set the assembler's location counter to the specified address (decimal or hex)
.WORD value: Place a 16-bit data word at the current address (can be a number or a label)

Number Formats
Decimal: 10, -5
Hexadecimal: 0x10, 0xFF
Binary: Not directly supported (use hex)
Labels can be used as operands and are resolved to their address (relative for jumps/branches, absolute for .WORD)

Assembly Program with RISKing: Example

```
START:
    LDI R1, 10           ; R1 <- 0x000A (decimal: 10)
    LDI R2, 5            ; R2 <- 0x0005 (decimal: 5)
    ADD R3, R1, R2       ; R3 <- R1 + R2 = 0x000A + 0x0005 = 0x000F
    SUB R4, R1, R2       ; R4 <- R1 - R2 = 0x000A - 0x0005 = 0x0005
    AND R5, R1, R2       ; R5 <- R1 & R2 = 0x000A & 0x0005 = 0x0000
    OR  R6, R1, R2       ; R6 <- R1 | R2 = 0x000A | 0x0005 = 0x000F
    HALT                 ; Stop CPU execution
```

```
; Register Allocation:
;  R1 (Operand A)   - First operand (value: 10)
;  R2 (Operand B)   - Second operand (value: 5)
;  R3 (Result ADD)  - Stores addition result
;  R4 (Result SUB)  - Stores subtraction result
;  R5 (Result AND)  - Stores bitwise AND result
;  R6 (Result OR)   - Stores bitwise OR result
```

```
; Expected Results:
;  R3 = 15 (0x0F) - Addition: 10 + 5
;  R4 = 5  (0x05) - Subtraction: 10 - 5
;  R5 = 0  (0x00) - AND: 10 & 5 (binary: 1010 & 0101 = 0000)
;  R6 = 15 (0x0F) - OR: 10 | 5 (binary: 1010 | 0101 = 1111)
```

Assembler: Convert Assembly to Binaries

- The RISKing assembler (RISKing_assembler.py) converts assembly code to binary format.
- Demo: Using RISKing_assembler
- Refer to **RISKing CPU Programming Manual.md**

```
START:
    LDI R1, 10
    LDI R2, 5
    ADD R3, R1, R2
    SUB R4, R1, R2
    AND R5, R1, R2
    OR  R6, R1, R2
    HALT
```

→

```
1000000100001010
1000001000000101
0000001100100010
0000110000100010
0010010100100010
0011011000100010
1111111111111111
```

Assembler: Convert Assembly to Binaries

Assembler Usage

The RISKing assembler (`RISKing_assembler.py`) converts assembly code to binary format.

Command Line

```
python RISKing_assembler.py input.asm [-o output_prefix] [-d memory_depth]
```

Parameters

- `input.asm`: Input assembly file
- `-o output_prefix`: Output file prefix (default: input filename)
- `-d memory_depth`: Memory depth in words (default: 1024)

Output Files

- `<prefix>.coe`: Vivado Block Memory initialization file (hex, COE format)
- `<prefix>.txt`: Binary format for simulation (`$readmemb`)
- `<prefix>.mem`: Memory initialization file (hex, one word per line)

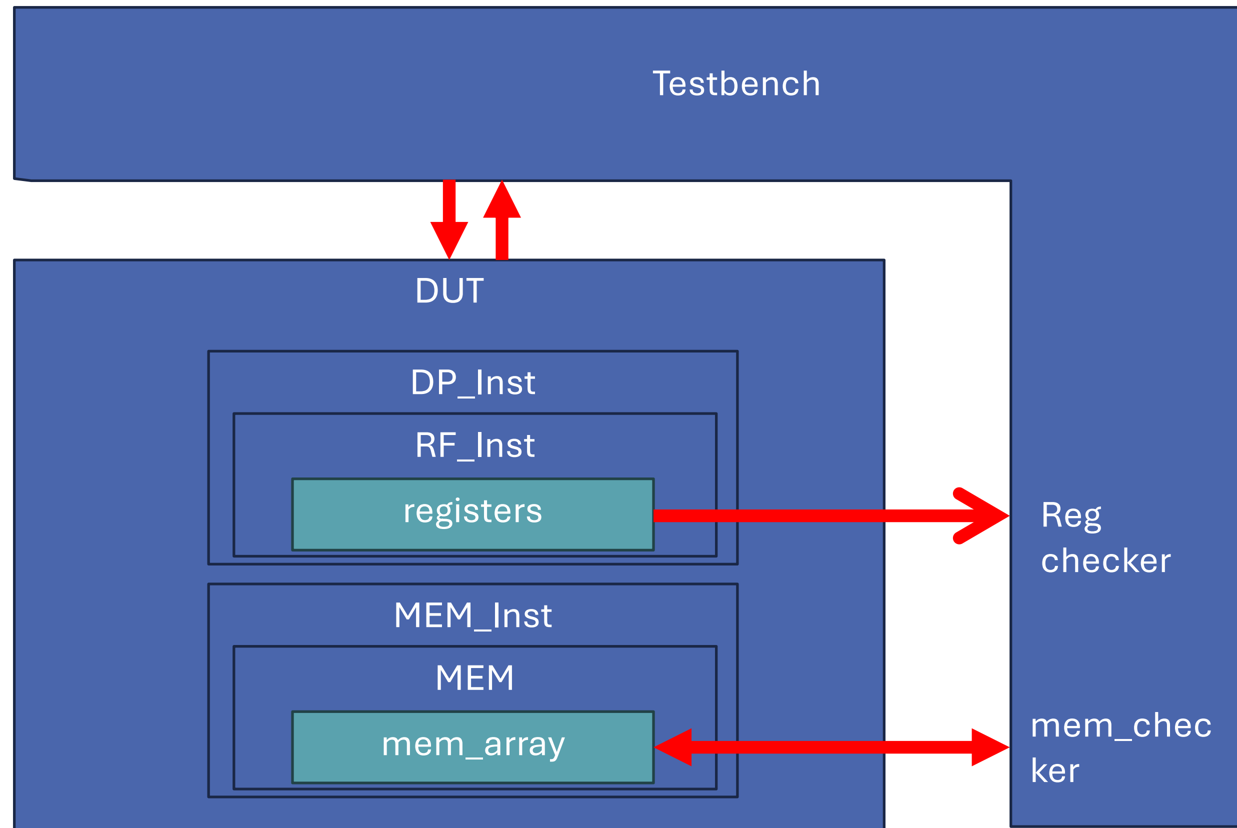
When `-o` is not specified, the output prefix defaults to `<input_basename>_mem_init` (e.g., `program.asm` → `program_mem_init.coe`, `program_mem_init.txt`, `program_mem_init.mem`).

Unused memory locations are filled with `0xFFFF` (HALT) up to the specified memory depth.

Example

```
python RISKing_assembler.py program.asm -o mem_init -d 1024
```

Code Evaluation: Testbench Hierarchy



Demo

- Finish CPU design
- Verify with Verilog Autograder With Waveforms
- Write your own assembly program
- Make a testbench with your program
- Download to BASYS3 for Demo