CS 211: Intro to Computer Architecture

10.1: RISC-V Assembly: Control Flow

Minesh Patel

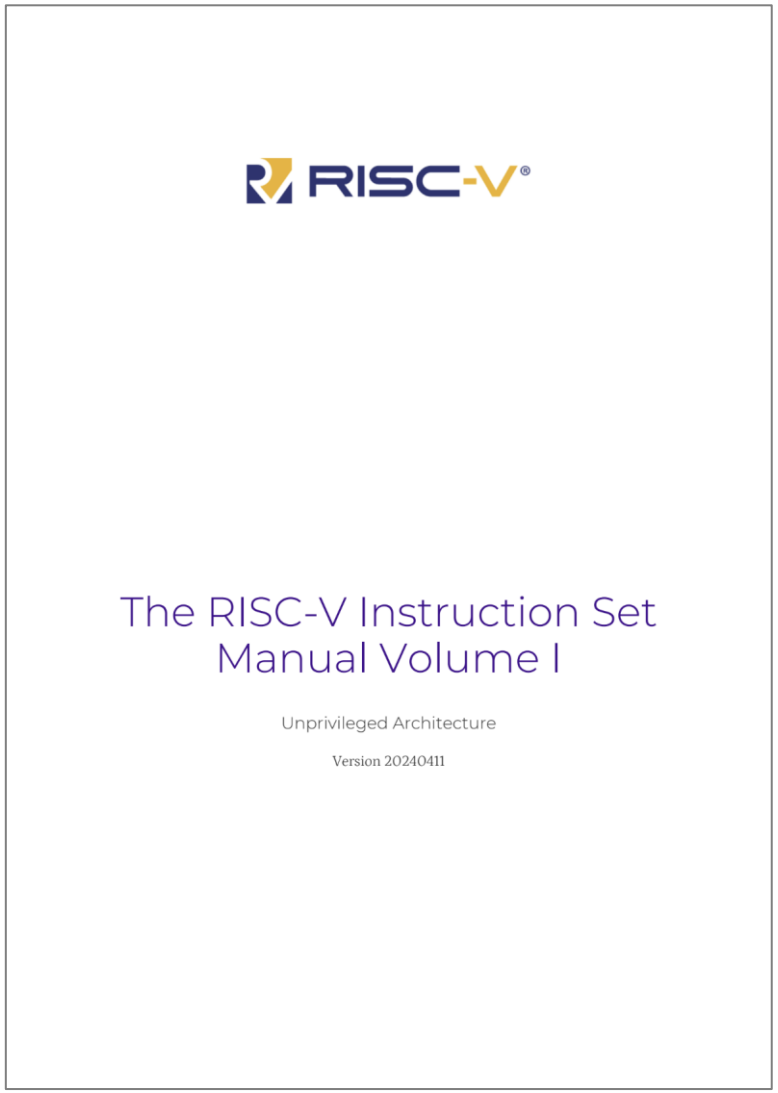
Spring 2025 – Tuesday 1 April

Announcements

- Assignments
  - **Extra Credit:** replaces WA6, due **Next Friday (April 11) @ 11:59 pm**

- Upcoming
  - **WA7:** released tonight
  - **PA4:** released tomorrow

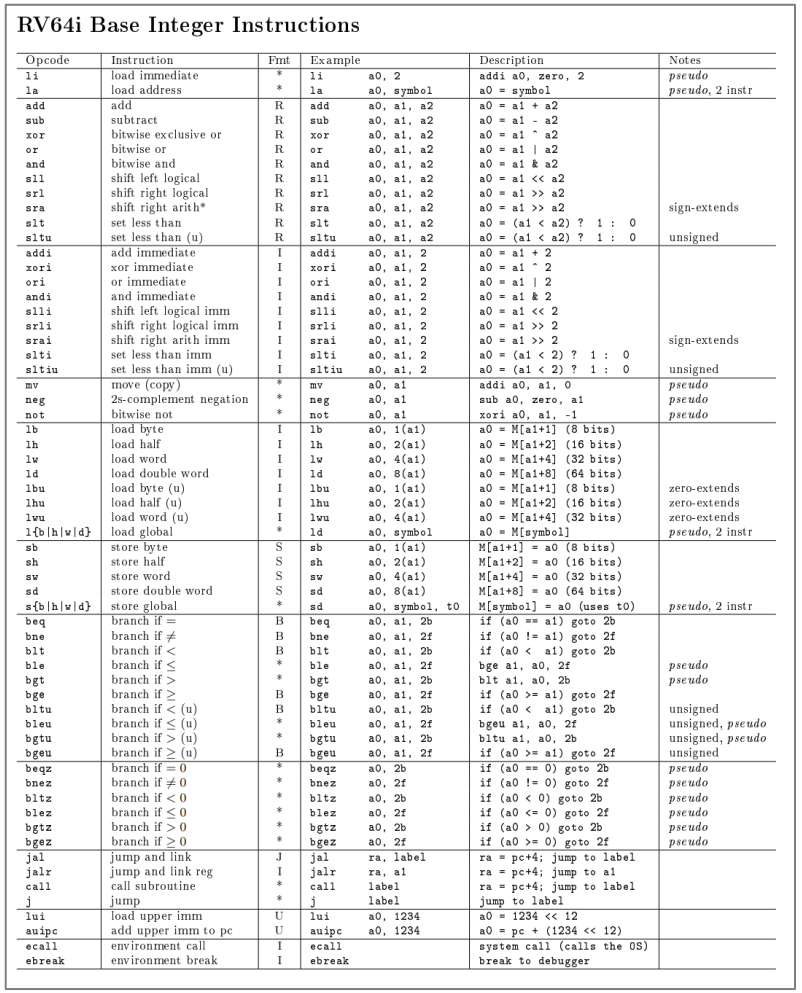
Useful Resources: Official Specifications



https://github.com/riscv/riscv-isa-manual/

https://github.com/riscv-non-isa/riscv-elf-psabi-doc

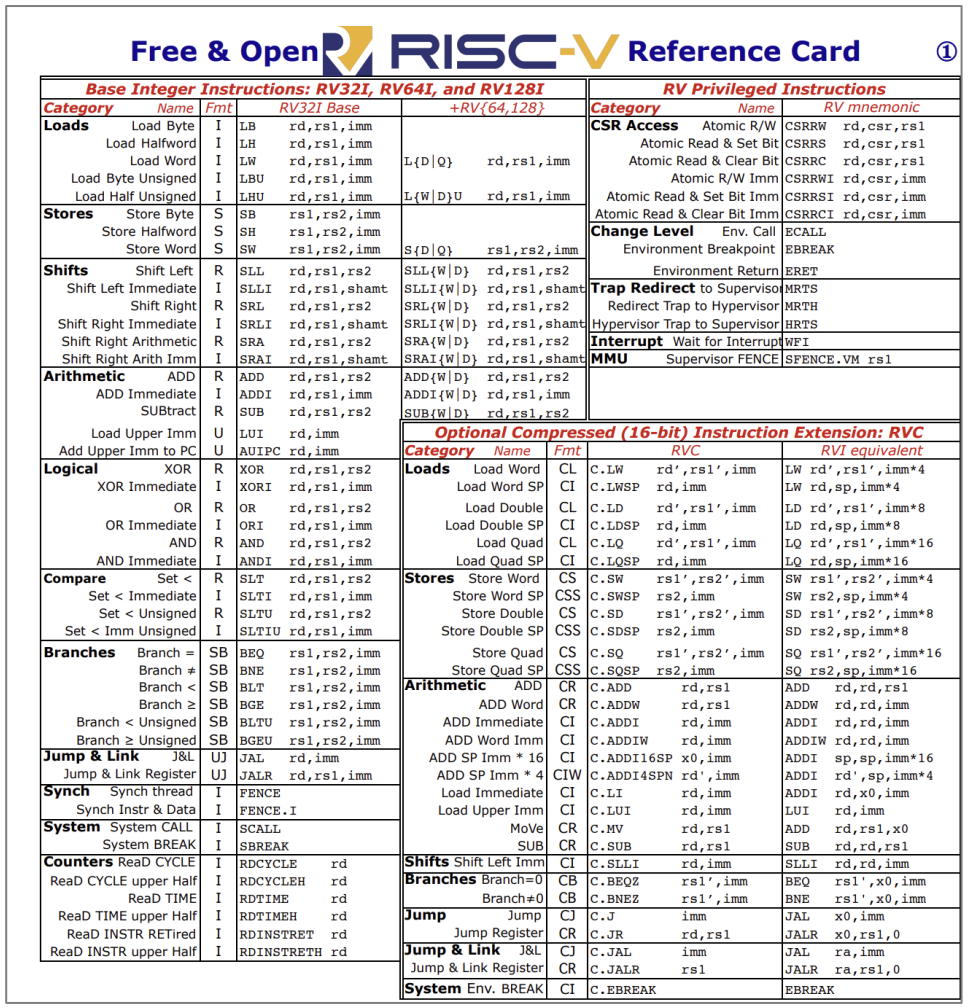
Useful Resources: Third Party ISA References



RV64i Base Integer Instructions

Opcode	Instruction	Fmt	Example		Description	Notes
li	load immediate	*	li	a0, 2	addi a0, zero, 2	*pseudo*
la	load address	*	la	a0, symbol	a0 = symbol	*pseudo*, 2 instr
add	add	R	add	a0, a1, a2	a0 = a1 + a2	
sub	subtract	R	sub	a0, a1, a2	a0 = a1 - a2	
xor	bitwise exclusive or	R	xor	a0, a1, a2	a0 = a1 ^ a2	
or	bitwise or	R	or	a0, a1, a2	a0 = a1 | a2	
and	bitwise and	R	and	a0, a1, a2	a0 = a1 & a2	
sll	shift left logical	R	sll	a0, a1, a2	a0 = a1 << a2	
srl	shift right logical	R	srl	a0, a1, a2	a0 = a1 >> a2	
sra	shift right arith*	R	sra	a0, a1, a2	a0 = a1 >> a2	sign-extends
slt	set less than	R	slt	a0, a1, a2	a0 = (a1 < a2) ? 1 : 0	
sltu	set less than (u)	R	sltu	a0, a1, a2	a0 = (a1 < a2) ? 1 : 0	unsigned
addi	add immediate	I	addi	a0, a1, 2	a0 = a1 + 2	
xori	xor immediate	I	xori	a0, a1, 2	a0 = a1 ^ 2	
ori	or immediate	I	ori	a0, a1, 2	a0 = a1 | 2	
andi	and immediate	I	andi	a0, a1, 2	a0 = a1 & 2	
slli	shift left logical imm	I	slli	a0, a1, 2	a0 = a1 << 2	
srli	shift right logical imm	I	srli	a0, a1, 2	a0 = a1 >> 2	
srai	shift right arith imm	I	srai	a0, a1, 2	a0 = a1 >> 2	sign-extends
slti	set less than imm	I	slti	a0, a1, 2	a0 = (a1 < 2) ? 1 : 0	
sltiu	set less than imm (u)	I	sltiu	a0, a1, 2	a0 = (a1 < 2) ? 1 : 0	unsigned
mv	move (copy)	*	mv	a0, a1	addi a0, a1, 0	*pseudo*
neg	2s-complement negation	*	neg	a0, a1	sub a0, zero, a1	*pseudo*
not	bitwise not	*	not	a0, a1	xori a0, a1, -1	*pseudo*
lb	load byte	I	lb	a0, 1(a1)	a0 = M[a1+1] (8 bits)	
lh	load half	I	lh	a0, 2(a1)	a0 = M[a1+2] (16 bits)	
lw	load word	I	lw	a0, 4(a1)	a0 = M[a1+4] (32 bits)	
ld	load double word	I	ld	a0, 8(a1)	a0 = M[a1+8] (64 bits)	
lbu	load byte (u)	I	lbu	a0, 1(a1)	a0 = M[a1+1] (8 bits)	zero-extends
lhu	load half (u)	I	lhu	a0, 2(a1)	a0 = M[a1+2] (16 bits)	zero-extends
lwu	load word (u)	I	lwu	a0, 4(a1)	a0 = M[a1+4] (32 bits)	zero-extends
l{b|h|w|d}	load global	*	ld	a0, symbol	a0 = M[symbol]	*pseudo*, 2 instr
sb	store byte	S	sb	a0, 1(a1)	M[a1+1] = a0 (8 bits)	
sh	store half	S	sh	a0, 2(a1)	M[a1+2] = a0 (16 bits)	
sw	store word	S	sw	a0, 4(a1)	M[a1+4] = a0 (32 bits)	
sd	store double word	S	sd	a0, 8(a1)	M[a1+8] = a0 (64 bits)	
s{b|h|w|d}	store global	*	sd	a0, symbol, t0	M[symbol] = a0 (uses t0)	*pseudo*, 2 instr
beq	branch if =	B	beq	a0, a1, 2b	if (a0 == a1) goto 2b	
bne	branch if ≠	B	bne	a0, a1, 2f	if (a0 != a1) goto 2f	
blt	branch if <	B	blt	a0, a1, 2b	if (a0 < a1) goto 2b	
ble	branch if ≤	*	ble	a0, a1, 2f	bge a1, a0, 2f	*pseudo*
bgt	branch if >	*	bgt	a0, a1, 2b	blt a1, a0, 2b	*pseudo*
bge	branch if ≥	B	bge	a0, a1, 2f	if (a0 >= a1) goto 2f	
bltu	branch if < (u)	B	bltu	a0, a1, 2b	if (a0 < a1) goto 2b	unsigned
bleu	branch if ≤ (u)	*	bleu	a0, a1, 2f	bgeu a1, a0, 2f	unsigned, *pseudo*
bgtu	branch if > (u)	*	bgtu	a0, a1, 2b	bltu a1, a0, 2b	unsigned, *pseudo*
bgeu	branch if ≥ (u)	B	bgeu	a0, a1, 2f	if (a0 >= a1) goto 2f	unsigned
beqz	branch if = 0	*	beqz	a0, 2b	if (a0 == 0) goto 2b	*pseudo*
bnez	branch if ≠ 0	*	bnez	a0, 2f	if (a0 != 0) goto 2f	*pseudo*
bltz	branch if < 0	*	bltz	a0, 2b	if (a0 < 0) goto 2b	*pseudo*
blez	branch if ≤ 0	*	blez	a0, 2f	if (a0 <= 0) goto 2f	*pseudo*
bgtz	branch if > 0	*	bgtz	a0, 2b	if (a0 > 0) goto 2b	*pseudo*
bgez	branch if ≥ 0	*	bgez	a0, 2f	if (a0 >= 0) goto 2f	*pseudo*
jal	jump and link	J	jal	ra, label	ra = pc+4; jump to label	
jalr	jump and link reg	I	jalr	ra, a1	ra = pc+4; jump to a1	
call	call subroutine	*	call	label	ra = pc+4; jump to label	
j	jump	*	j	label	jump to label	
lui	load upper imm	U	lui	a0, 1234	a0 = 1234 << 12	
auipc	add upper imm to pc	U	auipc	a0, 1234	a0 = pc + (1234 << 12)	
ecall	environment call	I	ecall		system call (calls the OS)	
ebreak	environment break	I	ebreak		break to debugger	

https://www.cs.utahtech.edu/cs/2810/riscv-card.pdf



Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	LB rd,rs1,imm	
	Load Halfword	I	LH rd,rs1,imm	
	Load Word	I	LW rd,rs1,imm	L{D|Q} rd,rs1,imm
	Load Byte Unsigned	I	LBU rd,rs1,imm	
	Load Half Unsigned	I	LHU rd,rs1,imm	L{W|D}U rd,rs1,imm
Stores	Store Byte	S	SB rs1,rs2,imm	
	Store Halfword	S	SH rs1,rs2,imm	
	Store Word	S	SW rs1,rs2,imm	S{D|Q} rs1,rs2,imm
Shifts	Shift Left	R	SLL rd,rs1,rs2	SLL{W|D} rd,rs1,rs2
	Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W|D} rd,rs1,shamt
	Shift Right	R	SRL rd,rs1,rs2	SRL{W|D} rd,rs1,rs2
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W|D} rd,rs1,shamt
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W|D} rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAI{W|D} rd,rs1,shamt
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADD{W|D} rd,rs1,rs2
	ADD Immediate	I	ADDI rd,rs1,imm	ADDI{W|D} rd,rs1,imm
	SUBtract	R	SUB rd,rs1,rs2	SUB{W|D} rd,rs1,rs2
	Load Upper Imm	U	LUI rd,imm	
	Add Upper Imm to PC	U	AUIPC rd,imm	
Logical	XOR	R	XOR rd,rs1,rs2	
	XOR Immediate	I	XORI rd,rs1,imm	
	OR	R	OR rd,rs1,rs2	
	OR Immediate	I	ORI rd,rs1,imm	
	AND	R	AND rd,rs1,rs2	
	AND Immediate	I	ANDI rd,rs1,imm	
Compare	Set <	R	SLT rd,rs1,rs2	
	Set < Immediate	I	SLTI rd,rs1,imm	
	Set < Unsigned	R	SLTU rd,rs1,rs2	
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches	Branch =	SB	BEQ rs1,rs2,imm	
	Branch ≠	SB	BNE rs1,rs2,imm	
	Branch <	SB	BLT rs1,rs2,imm	
	Branch ≥	SB	BGE rs1,rs2,imm	
	Branch < Unsigned	SB	BLTU rs1,rs2,imm	
	Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm	
Jump & Link	J&L	UJ	JAL rd,imm	
	Jump & Link Register	UJ	JALR rd,rs1,imm	
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
System	System CALL	I	SCALL	
	System BREAK	I	SBREAK	
Counters	ReaD CYCLE	I	RDCYCLE rd	
	ReaD CYCLE upper Half	I	RDCYCLEH rd	
	ReaD TIME	I	RDTIME rd	
	ReaD TIME upper Half	I	RDTIMEH rd	
	ReaD INSTR RETired	I	RDINSTRET rd	
	ReaD INSTR upper Half	I	RDINSTRETH rd	

RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	CSRRW rd,csr,rs1
	Atomic Read & Set Bit	CSRRS rd,csr,rs1
	Atomic Read & Clear Bit	CSRRC rd,csr,rs1
	Atomic R/W Imm	CSRRWI rd,csr,imm
	Atomic Read & Set Bit Imm	CSRRSI rd,csr,imm
	Atomic Read & Clear Bit Imm	CSRRCI rd,csr,imm
Change Level	Env. Call	ECALL
	Environment Breakpoint	EBREAK
	Environment Return	ERET
Trap Redirect	to Supervisor	MRTS
	Redirect Trap to Hypervisor	MRTH
	Hypervisor Trap to Supervisor	HRTS
Interrupt	Wait for Interrupt	WFI
MMU	Supervisor FENCE	SFENCE.VM rs1

Optional Compressed (16-bit) Instruction Extension: RVC

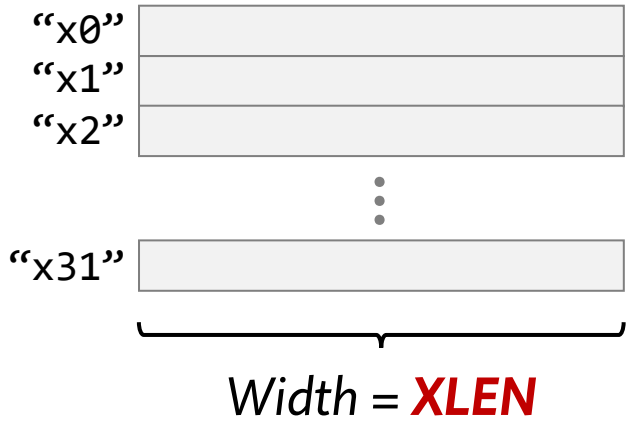
Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	C.LW rd',rs1',imm	LW rd',rs1',imm*4
	Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4
	Load Double	CL	C.LD rd',rs1',imm	LD rd',rs1',imm*8
	Load Double SP	CI	C.LDSP rd,imm	LD rd,sp,imm*8
	Load Quad	CL	C.LQ rd',rs1',imm	LQ rd',rs1',imm*16
	Load Quad SP	CI	C.LQSP rd,imm	LQ rd,sp,imm*16
Stores	Store Word	CS	C.SW rs1',rs2',imm	SW rs1',rs2',imm*4
	Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4
	Store Double	CS	C.SD rs1',rs2',imm	SD rs1',rs2',imm*8
	Store Double SP	CSS	C.SDSP rs2,imm	SD rs2,sp,imm*8
	Store Quad	CS	C.SQ rs1',rs2',imm	SQ rs1',rs2',imm*16
	Store Quad SP	CSS	C.SQSP rs2,imm	SQ rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1
	ADD Word	CR	C.ADDW rd,rs1	ADDW rd,rd,imm
	ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm
	ADD Word Imm	CI	C.ADDIW rd,imm	ADDIW rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm	ADDI rd',sp,imm*4
	Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm
	Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm
	MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0
	SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1
Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm
Branches	Branch=0	CB	C.BEQZ rs1',imm	BEQ rs1',x0,imm
	Branch≠0	CB	C.BNEZ rs1',imm	BNE rs1',x0,imm
Jump	Jump	CJ	C.J imm	JAL x0,imm
	Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0
Jump & Link	J&L	CJ	C.JAL imm	JAL ra,imm
	Jump & Link Register	CR	C.JALR rs1	JALR ra,rs1,0
System	Env. BREAK	CI	C.EBREAK	EBREAK

https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf

Correction: Word Size vs. XLEN

- RISC-V "word size" is **always 32-bits**
- Register width is "**XLEN**"

Registers



Load and Store Instructions

lb	load byte
lh	load half
lw	load word
ld	load double word
sb	store byte
sh	store half
sw	store word
sd	store double word

RISC-V Version	Register Width (XLEN)
RV32 (32-bit)	32-bits (single "word")
RV64 (64-bit)	64-bits (double "word")
RV128 (128-bit)	128-bits (quad "word")

Agenda

- **Executing a RISC-V Program**
  - **C to Machine Code**
  - The Program Counter
- Branch Instructions
  - Conditional Branches
  - C and ASM Control Flow

C to Machine Code

- Recall:
  - Function arguments in **a0-a10**
  - Return value **a0**

func.c

```c
#include <stdint.h>

uint64_t func(uint64_t *pa, uint64_t *pb)
{
    return *pa & *pb;
}
```

Compile to Assembly
(gcc –S)

func.S

```
func:
    ld a0, 0(a0)
    ld a5, 0(a1)
    and a0, a0, a5
    ret
```

Ilab Compilation Command

```
/common/users/shared/cs211_s25_5678/toolchai
n_glibc3/bin/riscv64-unknown-linux-gnu-gcc
-mabi=lp64 -march=rv64i
-Os
-S rv64i_func.c -o rv64i_func.o
```

Restricting ourselves to the RV64I ISA

C to Machine Code

func.c

```c
#include <stdint.h>

uint64_t func(uint64_t *pa, uint64_t *pb)
{
    return *pa & *pb;
}
```

Compile to asm
(gcc –S)

func.S

```
func:
    ld a0, 0(a0)
    ld a2, 0(a1)
    and a0, a0, a2
    ret
```

Assemble to object
(gcc –c)

func.o

```
0: 00053503
4: 0005b783
8: 00f57533
c: 00008067
```

```
mp2099@ilab1: ~/cs211/exper
mp2099@ilab1:~/cs211/experiment$ cat rv64i_func.c
#include <stdint.h>

uint64_t func(uint64_t *a, uint64_t *b)
{
    return *a & *b;
}
mp2099@ilab1:~/cs211/experiment$ /common/users/shared
/cs211_s25_5678/toolchain_glibc3/bin/riscv64-unknown-
linux-gnu-gcc -mabi=lp64 -march=rv64i -Os -c rv64i_fu
nc.c -o rv64i_func.o
mp2099@ilab1:~/cs211/experiment$ /common/users/shared
/cs211_s25_5678/toolchain_glibc3/bin/riscv64-unknown-
linux-gnu-objdump --section=.text -D rv64i_func.o

rv64i_func.o:     file format elf64-littleriscv


Disassembly of section .text:

0000000000000000 <func>:
   0:   00053503                ld      a0,0(a0)
   4:   0005b783                ld      a5,0(a1)
   8:   00f57533                and     a0,a0,a5
   c:   00008067                ret
mp2099@ilab1:~/cs211/experiment$
```

C to Machine Code

func.c

```c
#include <stdint.h>

uint64_t func(uint64_t *pa, uint64_t *pb)
{
    return *pa & *pb;
}
```

Compile to asm
(gcc –S)

func.S

```
func:
    ld a0, 0(a0)
    ld a2, 0(a1)
    and a0, a0, a2
    ret
```

Assemble to object
(gcc –c)

func.o

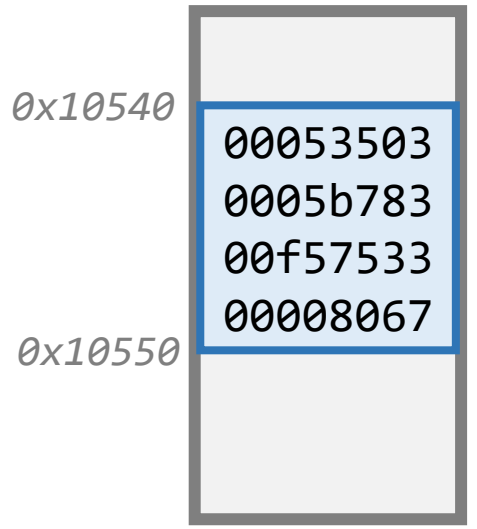
```
0: 00053503
4: 0005b783
8: 00f57533
c: 00008067
```

Link to executable
(gcc)

Disassembly of the Executable (func)

```
0000000000010540 <func>:
   10540:       00053503                ld      a0,0(a0)
   10544:       0005b783                ld      a5,0(a1)
   10548:       00f57533                and     a0,a0,a5
   1054c:       00008067                ret
```

Memory



Agenda

- Executing a RISC-V Program
  - C to Machine Code
  - **The Program Counter**
- Branch Instructions
  - Conditional Branches
  - C and ASM Control Flow

Program Counter (PC) and Instruction Register (IR)

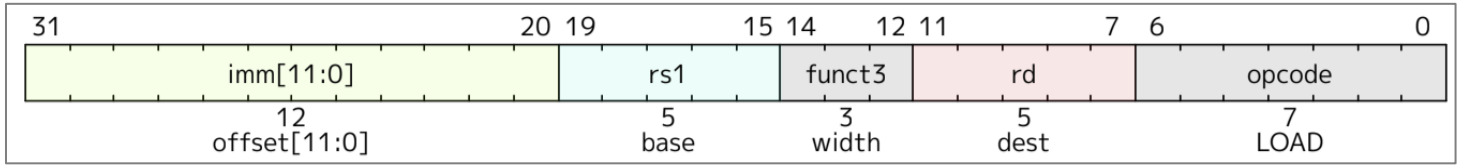
- The CPU "fetches" instructions from memory **in program order**
- Two special hardware registers:
  - **Program counter (PC)**: **pointer to** the current instruction
  - **Instruction register (IR)**: the current instruction's **number representation**

PC = 0x10540
next = PC + 4
...

```
0000000000010540 <func>:
   10540: IR =  00053503        ld      a0,0(a0)
   10544:       0005b783        ld      a5,0(a1)
   10548:       00f57533        and     a0,a0,a5
   1054c:       00008067        ret
```

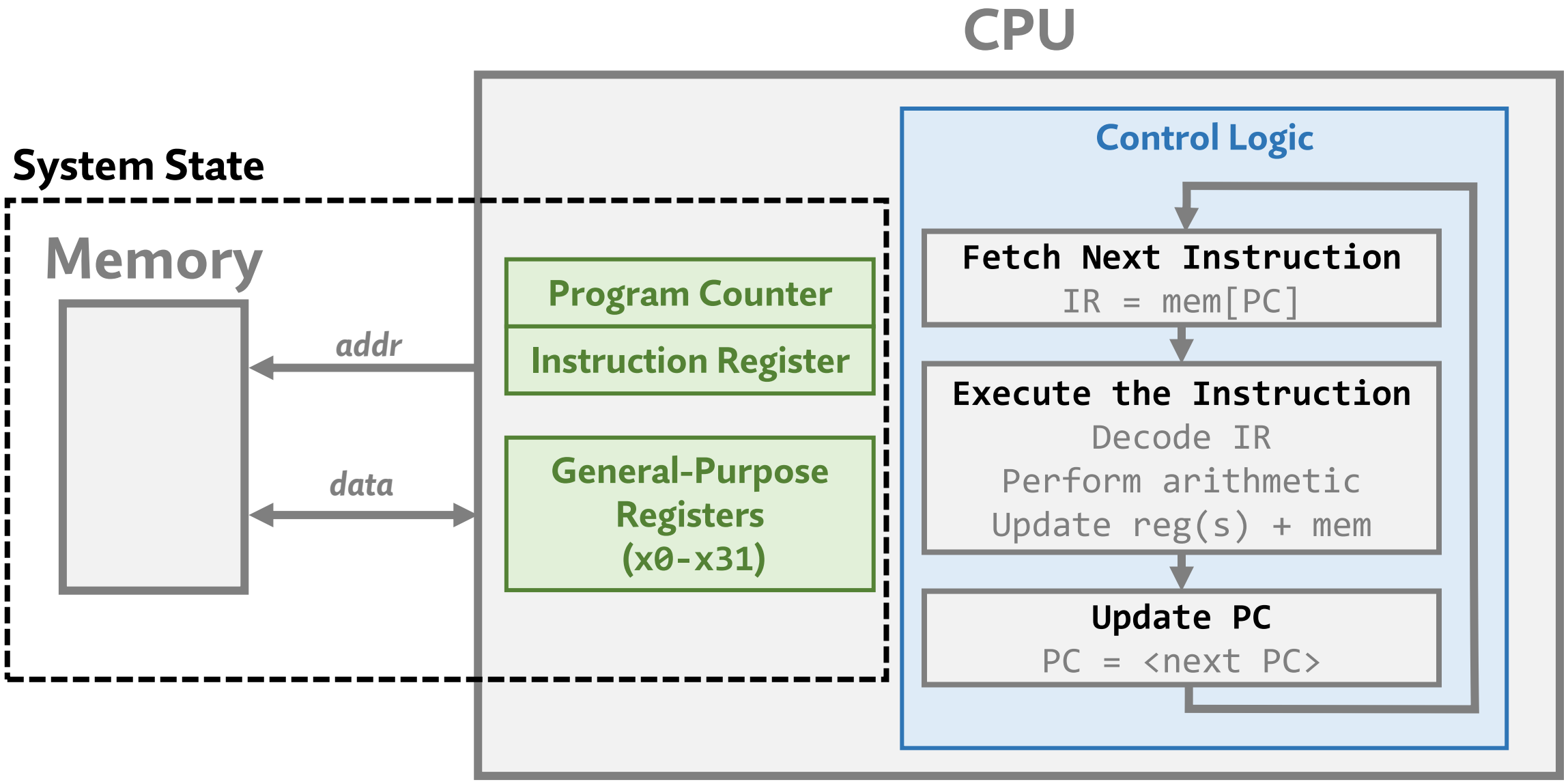
`IR = mem[PC] = 0x00053503`

Represents the machine language instruction "`ld a0, 0(a0)`"



"The RISC-V Instruction Set Manual Volume I", Version 20240411.

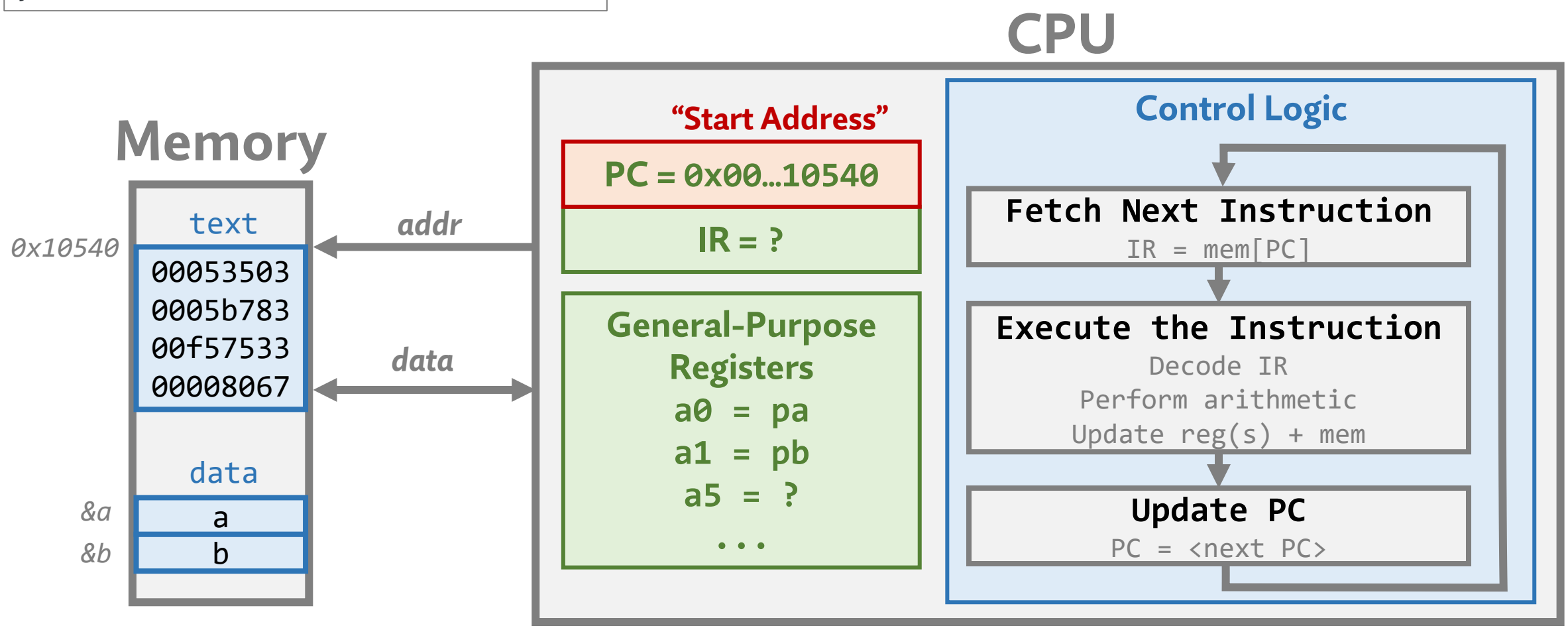
CPU Instruction Execution



Instruction Execution: Initial State

```c
#include <stdint.h>

uint64_t func(uint64_t *pa, uint64_t *pb)
{
    return *pa & *pb;
}
```

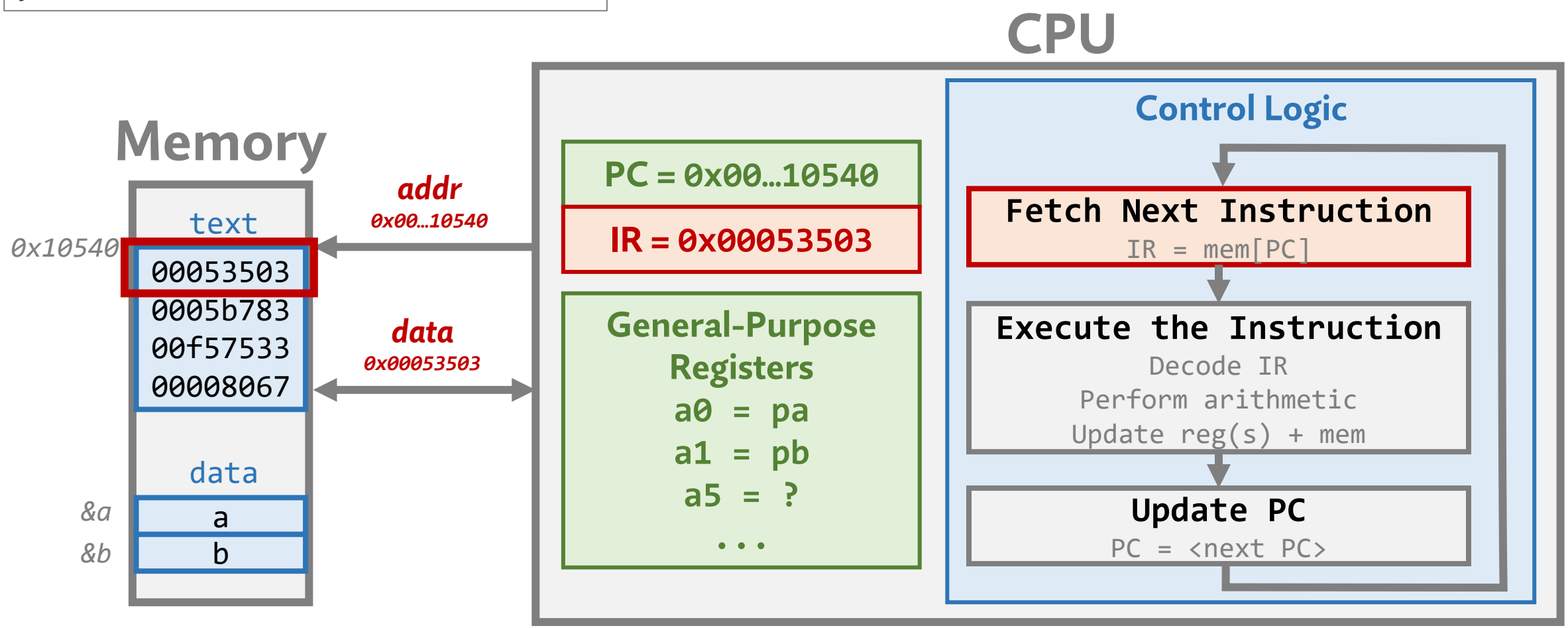
```
0000000000010540 <func>:
   10540:       00053503                ld      a0,0(a0)
   10544:       0005b783                ld      a5,0(a1)
   10548:       00f57533                and     a0,a0,a5
   1054c:       00008067                ret
```



Instruction Execution (1/3): Instruction Fetch

```c
#include <stdint.h>

uint64_t func(uint64_t *pa, uint64_t *pb)
{
    return *pa & *pb;
}
```

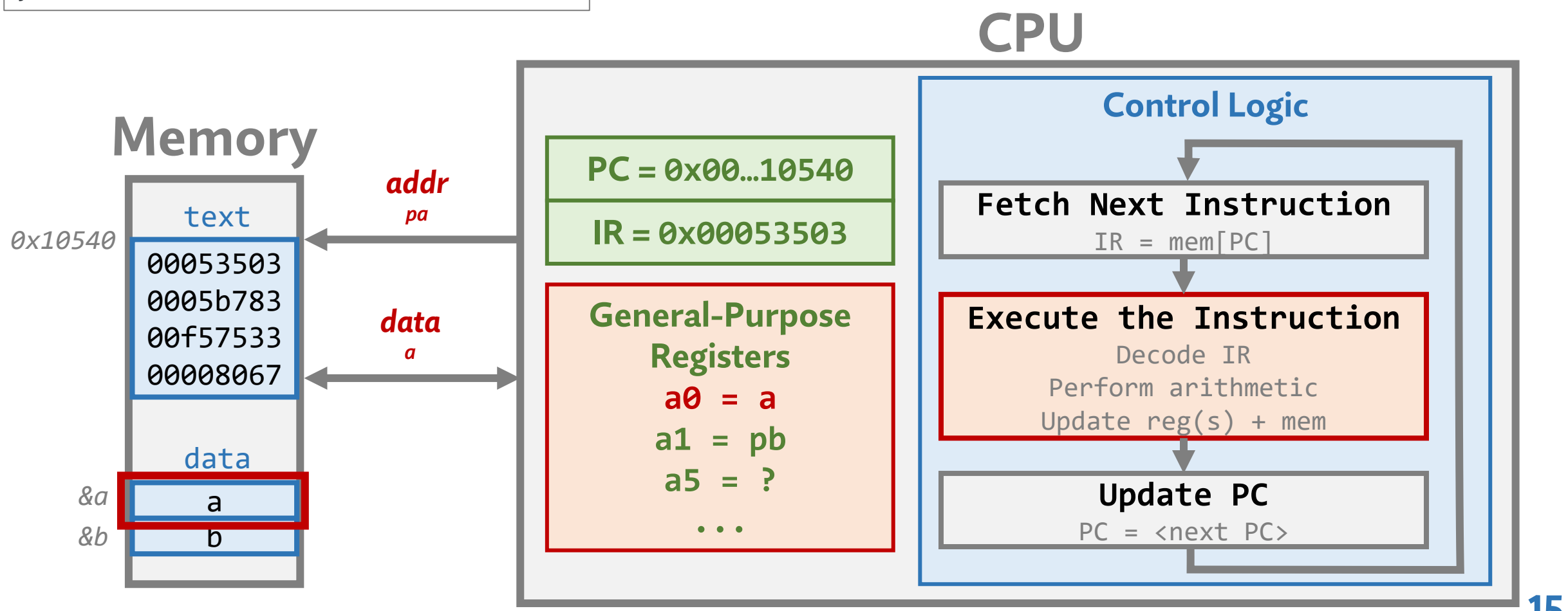
```
0000000000010540 <func>:
   10540:       00053503                ld      a0,0(a0)
   10544:       0005b783                ld      a5,0(a1)
   10548:       00f57533                and     a0,a0,a5
   1054c:       00008067                ret
```



Instruction Execution (2/3): Instruction Execution

```c
#include <stdint.h>

uint64_t func(uint64_t *pa, uint64_t *pb)
{
    return *pa & *pb;
}
```

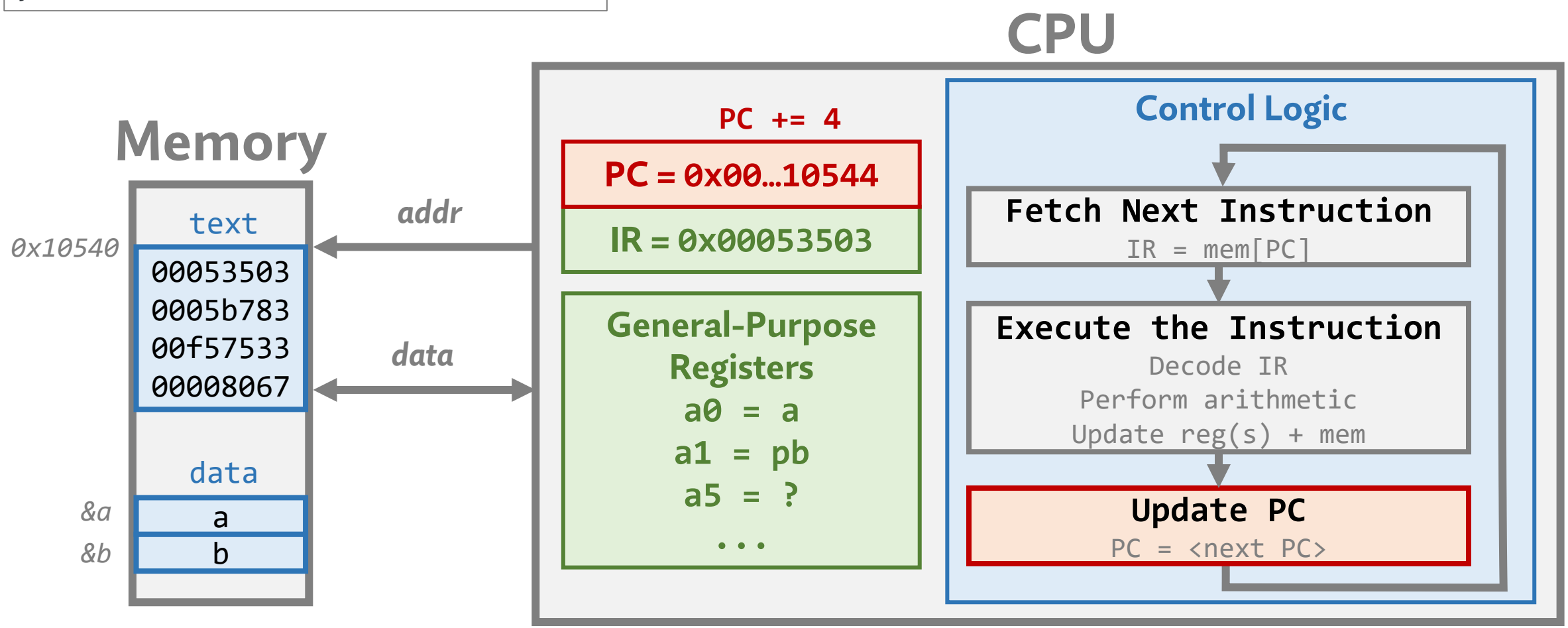
```
0000000000010540 <func>:
   10540:       00053503                ld      a0,0(a0)
   10544:       0005b783                ld      a5,0(a1)
   10548:       00f57533                and     a0,a0,a5
   1054c:       00008067                ret
```



Instruction Execution (3/3): Update PC

```c
#include <stdint.h>

uint64_t func(uint64_t *pa, uint64_t *pb)
{
    return *pa & *pb;
}
```

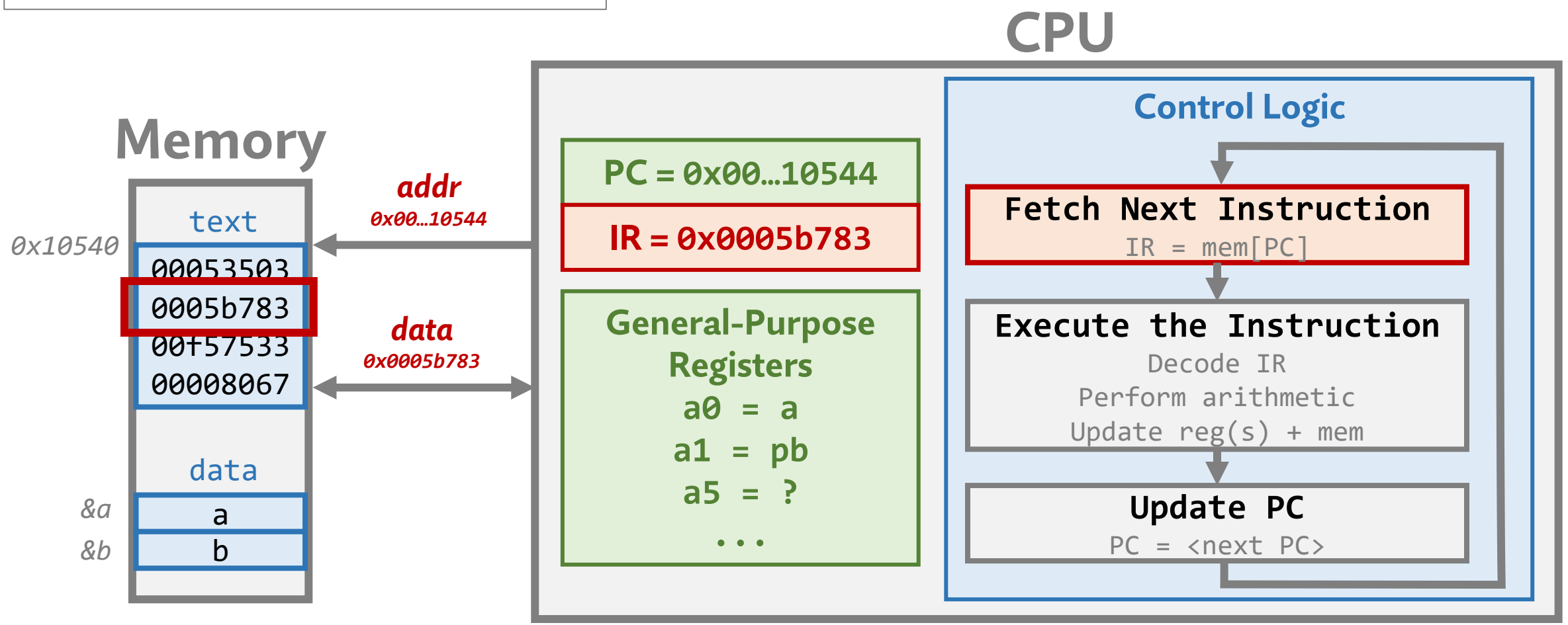
```
0000000000010540 <func>:
   10540:       00053503                ld      a0,0(a0)
   10544:       0005b783                ld      a5,0(a1)
   10548:       00f57533                and     a0,a0,a5
   1054c:       00008067                ret
```



Instruction Execution: Next Instruction Fetch

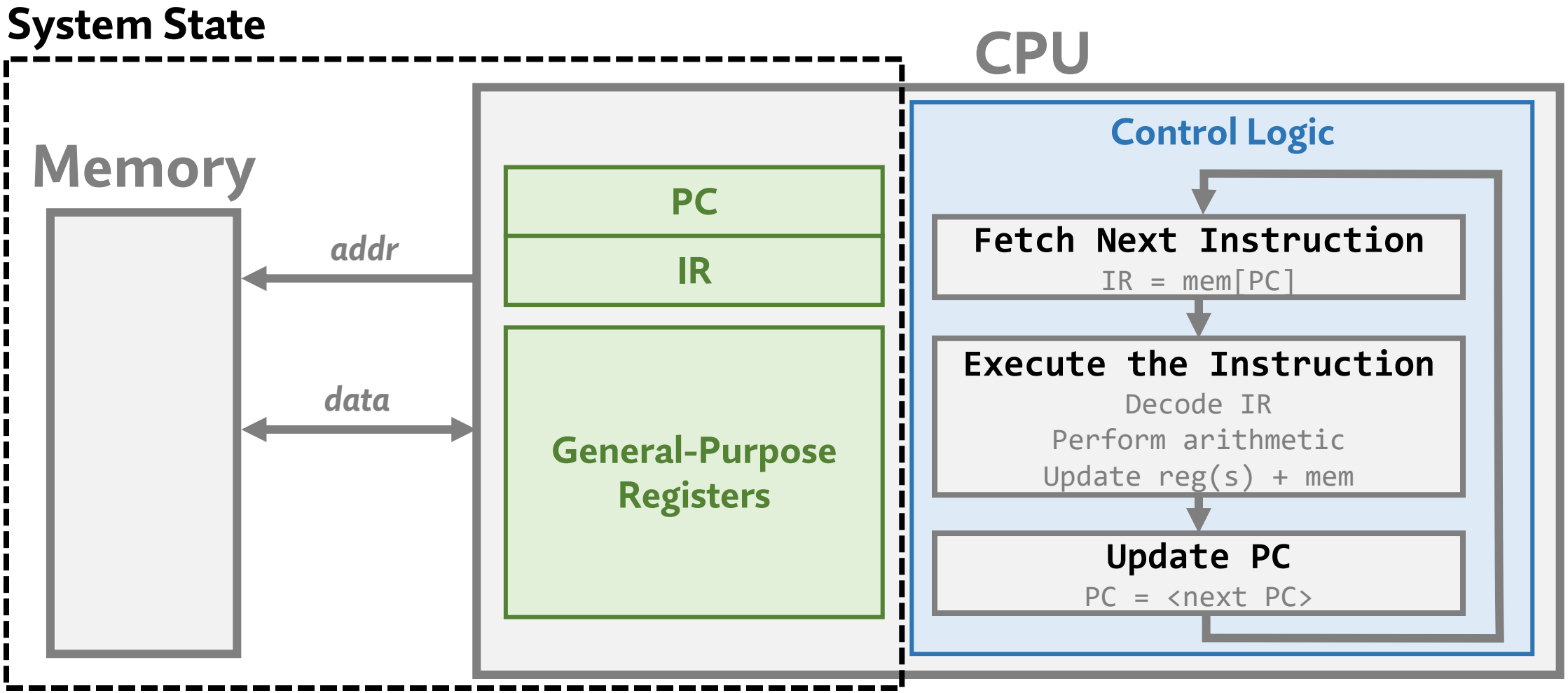
```c
#include <stdint.h>

uint64_t func(uint64_t *pa, uint64_t *pb)
{
    return *pa & *pb;
}
```

```
0000000000010540 <func>:
   10540:       00053503                ld      a0,0(a0)
   10544:       0005b783                ld      a5,0(a1)
   10548:       00f57533                and     a0,a0,a5
   1054c:       00008067                ret
```



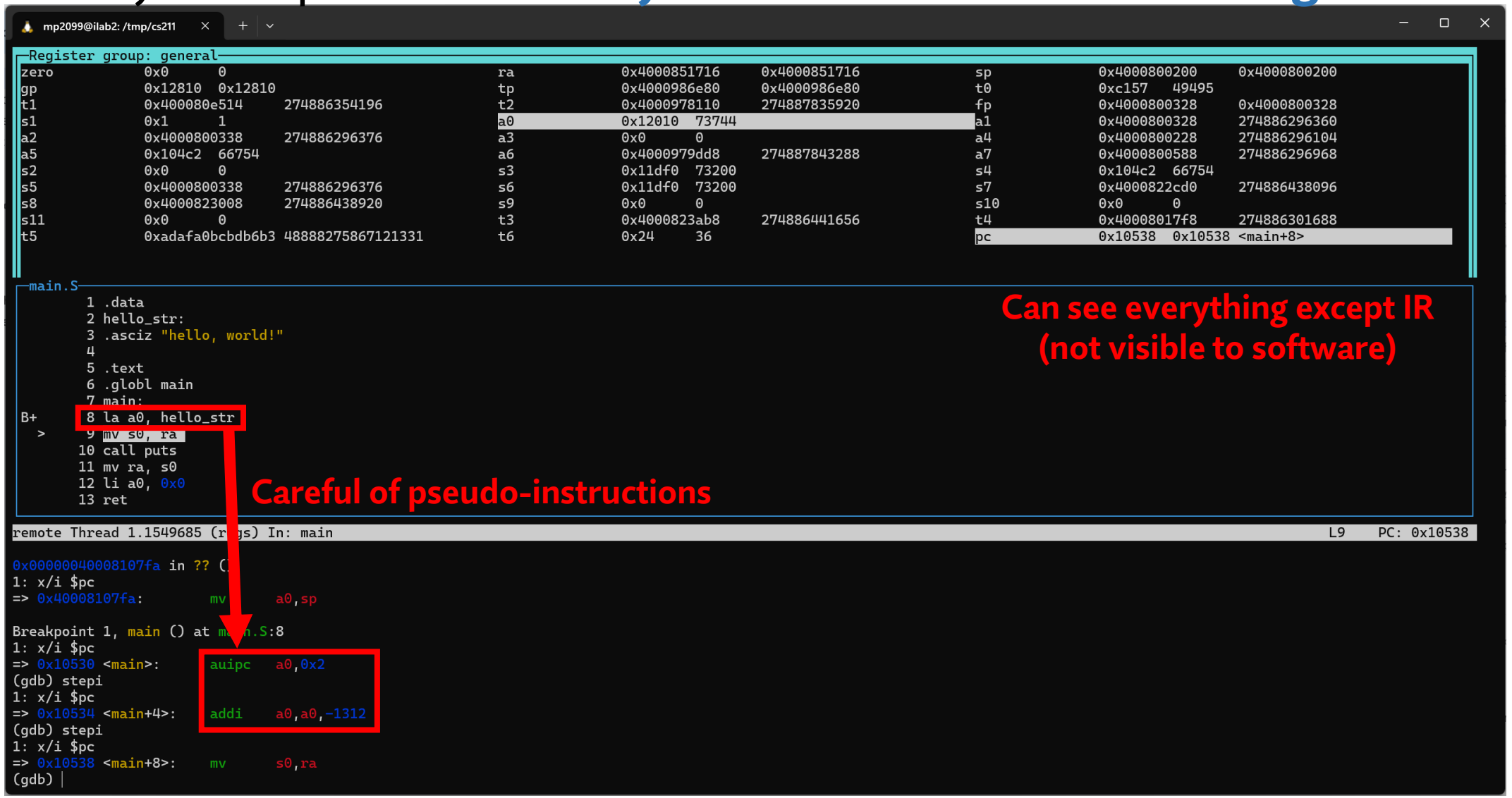
Overview: Instruction Execution

- **Program:** Sequence of instructions that **reads/modifies the system state**



Inspecting System State with GDB

- GDB lets you step **instruction by instruction** and **monitor registers**



Commands for the Previous GDB Example

```
.data
hello_str:
      .asciz "hello, world!"

.text
.globl main
main:
      la a0, hello_str
      mv s0, ra
      call puts
      mv ra, s0
      li a0, 0x0
      ret
```

main.S

Terminal 1

```
mp2099@ilab1:~/cs211/experiment$ /common/users/shared/cs211_s25_5678/toolchain_glibc3/bin/riscv64-unknown-linux-gnu-gcc -Wl,-rpath=/common/users/shared/cs211_s25_5678/toolchain_glibc3/sysroot/lib -Wl,--dynamic-linker=/common/users/shared/cs211_s25_5678/toolchain_glibc3/sysroot/lib/ld-linux-riscv64-lp64.so.1 -g -march=rv64i -mabi=lp64 main.S -o main
mp2099@ilab1:~/cs211/experiment$ /common/system/riscv64i/grun main
```

Terminal 2

```
mp2099@ilab1:~/cs211/experiment$ /common/system/riscv64i/gdb main
```

Agenda

- Executing a RISC-V Program
  - C to Machine Code
  - The Program Counter
- **Branch Instructions**
  - Conditional Branches
  - C and ASM Control Flow

Calculating the Next PC

- **So far:** next instruction was always **PC + 4**
  - Called "straight-line code" or "basic block"
- **Consider:** What if the next instruction is **somewhere else**?

Example: If/Else

```
uint64_t func(uint64_t a)
{
    if(a == 0)
        return 34;
    else
        return 86;
}
```

a == 0

a != 0

Conditional Branch

Example: Function Call

```
uint64_t func(uint64_t a)
{
    ...
}

int main()
{
    func(0);
}
```

PC - ???

Unconditional Branch = "Jump"

Agenda

- Executing a RISC-V Program
  - C to Machine Code
  - The Program Counter
- Branch Instructions
  - **Conditional Branches**
  - C and ASM Control Flow

Conditional Branch Instructions

func.c

```
uint64_t func(uint64_t a)
{
    if(a == 0)
        return 34;
    else
        return 86;
}
```

a == 0

a != 0

beq = branch "if equal"

func.S

```
if(a0 == x0)
    PC += 4
else
    PC += 12
```

```
func:
    beq a0, x0, ret_34
    li a0, 86
    ret
ret_34:
    li a0, 34
    ret
```

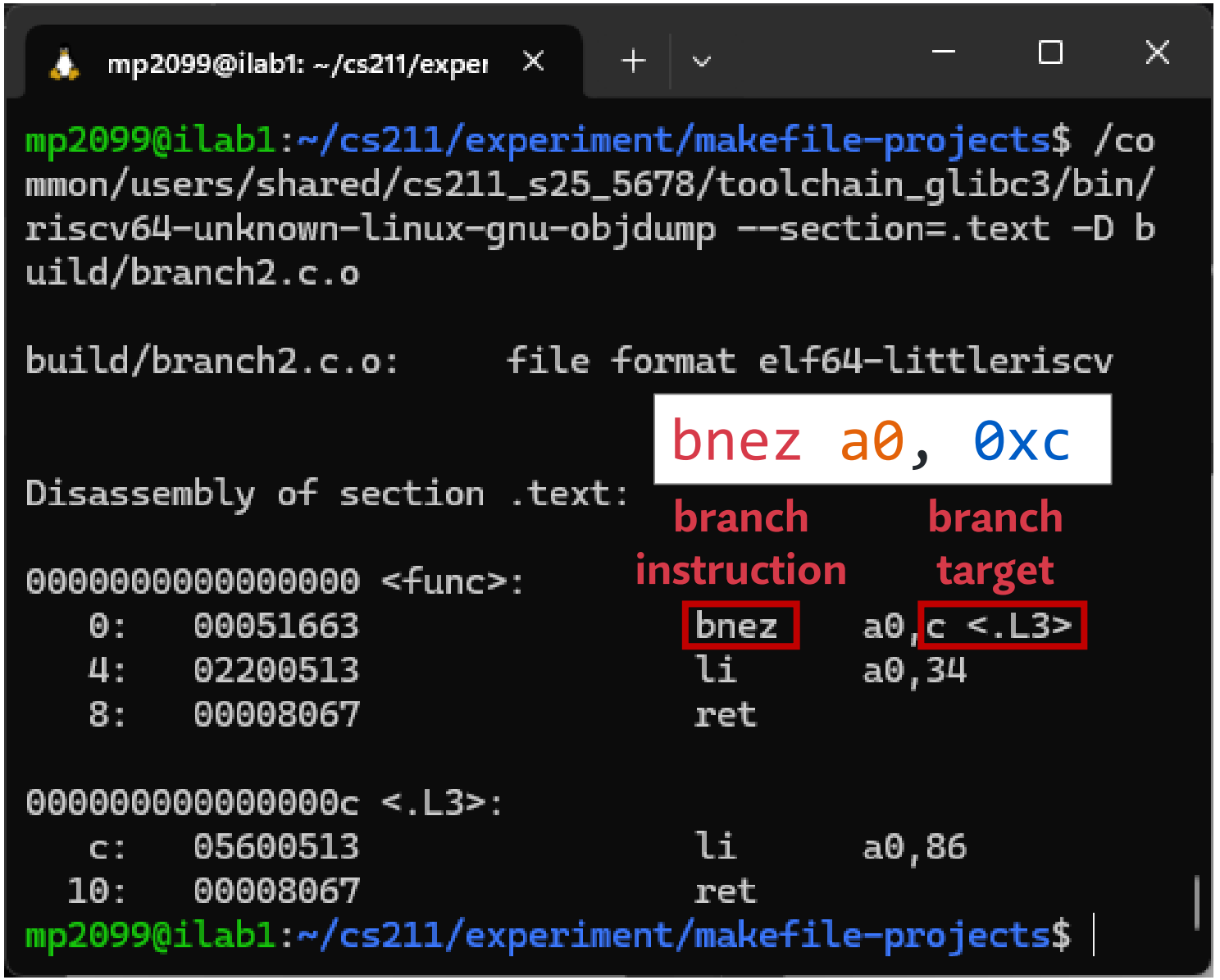
```
b<cond> xN, <PC offset>
```

- b<cond>: comparison (==, !=, <, <=, >, >=)
- xN: destination register
- <PC offset>: where to go if branch is taken

Example: pseudo-op bnez

```
uint64_t func(uint64_t a)
{
    if(a == 0)
        return 34;
    else
        return 86;
}
```

```
// bnez pseudocode
if(!a0)
    PC += 0xc;
else
    PC += 0x4;
```



Conditional Branch Instructions

```
b<cond> xN, <PC offset>
```

branch condition

destination register

where to go if branch is taken

beq	branch if =	B	beq	a0, a1, 2b	if (a0 == a1) goto 2b	
bne	branch if ≠	B	bne	a0, a1, 2f	if (a0 != a1) goto 2f	
blt	branch if <	B	blt	a0, a1, 2b	if (a0 < a1) goto 2b	
ble	branch if ≤	*	ble	a0, a1, 2f	bge a1, a0, 2f	*pseudo*
bgt	branch if >	*	bgt	a0, a1, 2b	blt a1, a0, 2b	*pseudo*
bge	branch if ≥	B	bge	a0, a1, 2f	if (a0 >= a1) goto 2f	
bltu	branch if < (u)	B	bltu	a0, a1, 2b	if (a0 < a1) goto 2b	unsigned
bleu	branch if ≤ (u)	*	bleu	a0, a1, 2f	bgeu a1, a0, 2f	unsigned, *pseudo*
bgtu	branch if > (u)	*	bgtu	a0, a1, 2b	bltu a1, a0, 2b	unsigned, *pseudo*
bgeu	branch if ≥ (u)	B	bgeu	a0, a1, 2f	if (a0 >= a1) goto 2f	unsigned
beqz	branch if = 0	*	beqz	a0, 2b	if (a0 == 0) goto 2b	*pseudo*
bnez	branch if ≠ 0	*	bnez	a0, 2f	if (a0 != 0) goto 2f	*pseudo*
bltz	branch if < 0	*	bltz	a0, 2b	if (a0 < 0) goto 2b	*pseudo*
blez	branch if ≤ 0	*	blez	a0, 2f	if (a0 <= 0) goto 2f	*pseudo*
bgtz	branch if > 0	*	bgtz	a0, 2b	if (a0 > 0) goto 2b	*pseudo*
bgez	branch if > 0	*	bgez	a0, 2f	if (a0 >= 0) goto 2f	*pseudo*

https://www.cs.utahtech.edu/cs/2810/riscv-card.pdf

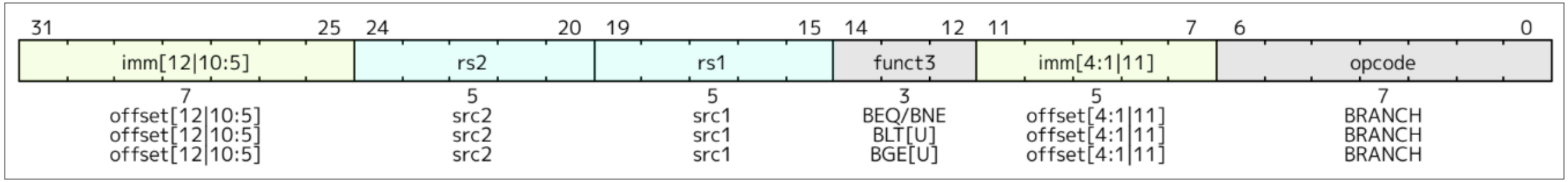
PC-Relative Addressing

```
b<cond> xN, <PC offset>
```

where to go if branch is taken

- Branch target is **relative** to the current PC
  - If branch NOT taken: **`PC = PC + 4`**
  - If branch taken: **`PC = PC + sext(offset[11:0] << 1)`**

- **Limitation:** the branch target must be within ±4 KiB of the branch's PC



https://github.com/riscv/riscv-isa-manual/

Specifying Branch Targets in Assembly

- We rarely calculate PC offsets by hand: in assembly, we use **labels**

func.c

```
void func(void)
{
    while(1) {}
}
```

func.S

```
func:
    beqz x0, func
```

func.o

```
0:  00000063
```

func.c

```
uint64_t func(uint64_t a)
{
    if(a == 0)
        return 34;
    else
        return 86;
}
```

func.S

```
func:
    bnez a0, .label
    li a0, 34
    ret

.label:
    li a0, 86
    ret
```

func.o

```
0:  00051663
4:  02200513
8:  00008067
c:  05600513
10: 00008067
```

Labels in C and Assembly

- C and assembly both support **labels** in (nearly) the same way

func.c

```
uint64_t func(uint64_t a)
{
    if(a == 0)
        return 34;
    else
        return 86;
}
```

func_goto.c

```
uint64_t func(uint64_t a)
{
    if(a == 0)
        goto btarget;
    return 34;

btarget:
    return 86;
}
```

func.S

```
func:
    beqz a0, .btarget
    li a0, 86
    ret

.btarget:
    li a0, 34
    ret
```

goto is unpopular in C
(but it exists)

branch instructions
are essentially **if+goto**

Agenda

- Executing a RISC-V Program
  - C to Machine Code
  - The Program Counter
- Branch Instructions
  - Conditional Branches
  - **C and ASM Control Flow**

Program Control Flow (C vs. Assembly)

- C has many control flow statements
  - **Conditionals:** if, else, ternary operator
  - **Loops:** while, for, do-while
  - **Switch/case statements**
  - **Break/continue/goto**
- Assembly has only **branches**
  - Essentially **if + goto**

C-Style Code

```
uint64_t func(uint64_t a)
{
    if(a == 0)
        return 34;
    else
        return 86;
}
```

Translation

ASM-Style Pseudo-C

```
uint64_t func(uint64_t a)
{
    if(a == 0)
        goto label;
    return 86;

label:
    return 34;
}
```

Translating C to Assembly

C Object	Register
uint64_t a	a0
Return value	a0

- General algorithm:

1. Convert all control flow statements to **if + goto**
2. Map all C objects to registers (i.e., *register allocation*)
3. Translate to assembly line-by-line

C-Style Code

```
uint64_t func(uint64_t a)
{
    if(a == 0)
        return 34;
    else
        return 86;
}
```

ASM-Style Pseudo-C

```
uint64_t func(uint64_t a)
{
    if(a == 0)
        goto label;
    return 86;

label:
    return 34;
}
```

ASM

```
func:
    beqz a0, .label
    li a0, 86
    ret

.label:
    li a0, 34
    ret
```

Generalizing If… else…

- Can define as many labels as needed

C-Style Code

```
uint64_t func(int64_t a)
{
    if(a == 0)
        puts("a == 0");
    else if(a < 0)
        puts("a < 0") ;
    else
        puts("a > 0") ;
    return 42;
}
```

ASM-Style Pseudo-C

```
uint64_t func(uint64_t a)
{
    if(a == 0)
        goto func_eqz;
    if(a < 0)
        goto func_ltz;
    puts("a != 0");
    goto common;

func_eqz:
    puts("a == 0");
    goto common;

func_ltz:
    puts("a < 0");

common:
    return 42;
}
```

Implementing Loops: do {…} while(…)

- Loops are also implemented with **branches and labels**

C-Style Code

```
void func(uint64_t *i)
{
    do
    {
        (*i)++;
    } while(*i < 10);
}
```

ASM-Style Pseudo-C

```
void func(uint64_t *i)
{
loop:
    (*i)++;
    if(*i < 10)
        goto loop;
}
```

Implementing Loops: `while(…)`

- While loops typically require an extra label

C-Style Code

```
void func(uint64_t *i)
{
    while(*i < 10)
    {
        (*i)++;
    }
}
```

ASM-Style Pseudo-C

```
void func(uint64_t *i)
{
loop:
    if(*i >= 10)
        goto done;
    (*i)++;
    goto loop;

done:
    return;
}
```

Implementing Loops: for(…)

- Just turn **for** into **while**

C-Style Code

```
void func(uint64_t i)
{
    for(uint64_t i = 0; i < 10; i++)
    {
        ...
    }
}
```

ASM-Style Pseudo-C

```
void func(void)
{
    uint64_t i;

loop:
    if(i >= 10)
        goto done;
    i++;
    goto loop;

done:
    return;
}
```

Conditionals Without Branches

- Compare two numbers: "**set less than**"

```
slt(u) rd, rs0, rs1        slti(u) rd, rs0, imm
```

```
rd = rs1 < rs2 ? 1 : 0;
```

Pseudocode

```
rd = rs1 < imm ? 1 : 0;
```

- Enables **avoiding branches** (among other uses)

C Code

```
void func(void)
{
    if(a0 < a1)
        a2++;
    ...
}
```

ASM

```
func:
    slt t0, a0, a1
    addi a2, a2, t0
    ...
```

RV64i So Far

Instructions

Arithmetic

```
add(i) rd, rs1, rs2/imm
sub(i) rd, rs1, rs2/imm
and(i) rd, rs1, rs2/imm
or(i) rd, rs1, rs2/imm
xor(i) rd, rs1, rs2/imm
sll(i) rd, rs1, rs2/imm
srl(i) rd, rs1, rs2/imm
sra(i) rd, rs1, rs2/imm
```

Comparison

```
slt(u)(i) rd, rs1, rs1/imm
```

Loads/Stores

```
l{b/h/w/d} rd, offset(rs)
lu{b/h/w/d} rd, offset(rs)
s{b/h/w/d} rd, offset(rs)
```

Branches

```
beq rs1, rs2, <label>
bne rs1, rs2, <label>
blt(u) rs1, rs2, <label>
bge(u) rs1, rs2, <label>
```

Pseudo-Instructions

Constants/Initialization

```
.asciz <C-style string>
mv rd, rs1
li rd, imm
la rd, <label>
```

Branches

```
ble(u) rd, <label>
bgt(u) rd, <label>
b{cond}z rd, <label>
```

CS 211: Intro to Computer Architecture

10.1: RISC-V Assembly: Control Flow

Minesh Patel

Spring 2025 – Tuesday 1 April