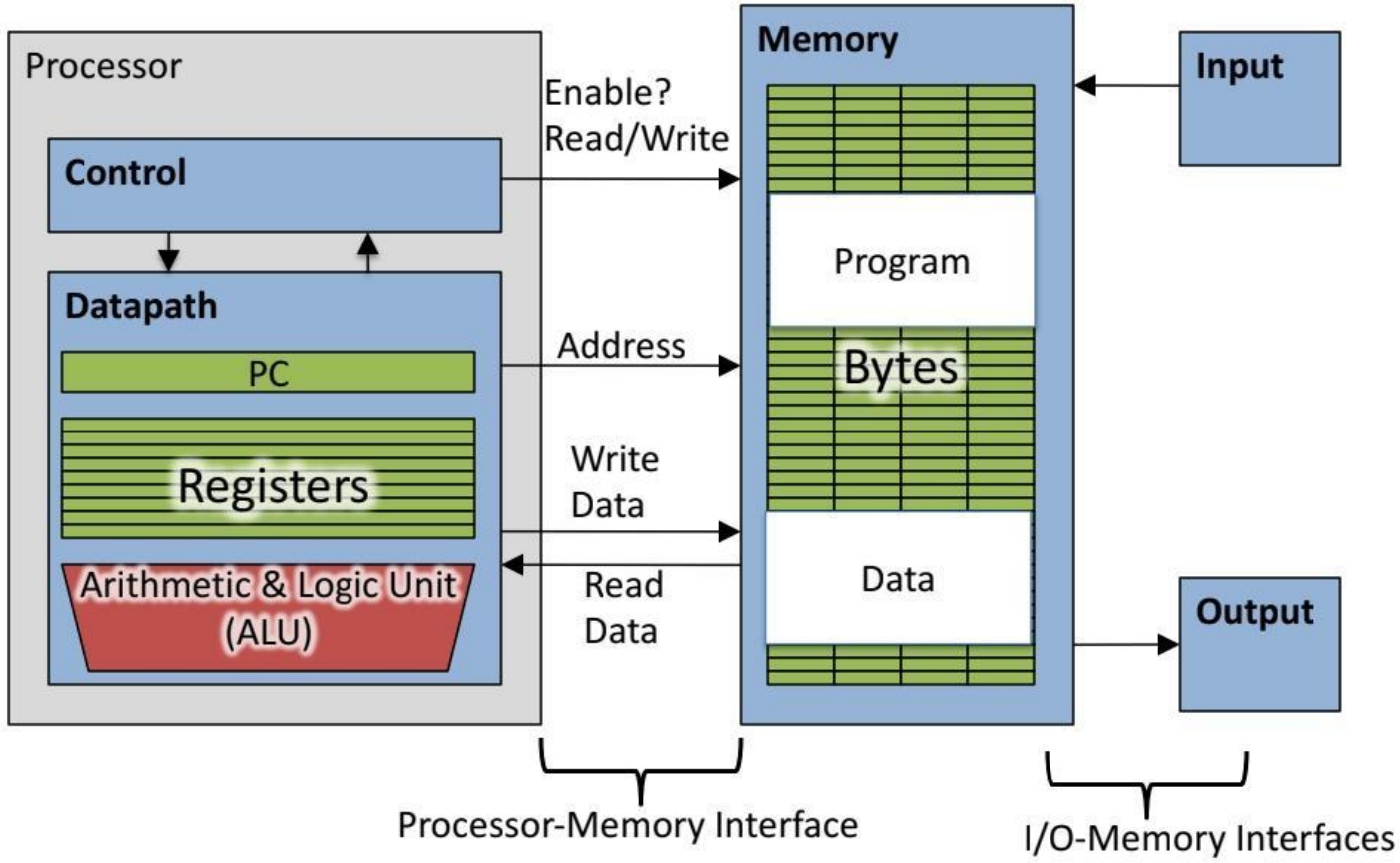
RISC V Assembly Programs

Dr. Girish H
Professor
Department of ECE
Cambridge Institute of Technology
&
Kavinesh
Research Staff
CCCIR
Cambridge Institute of Technology

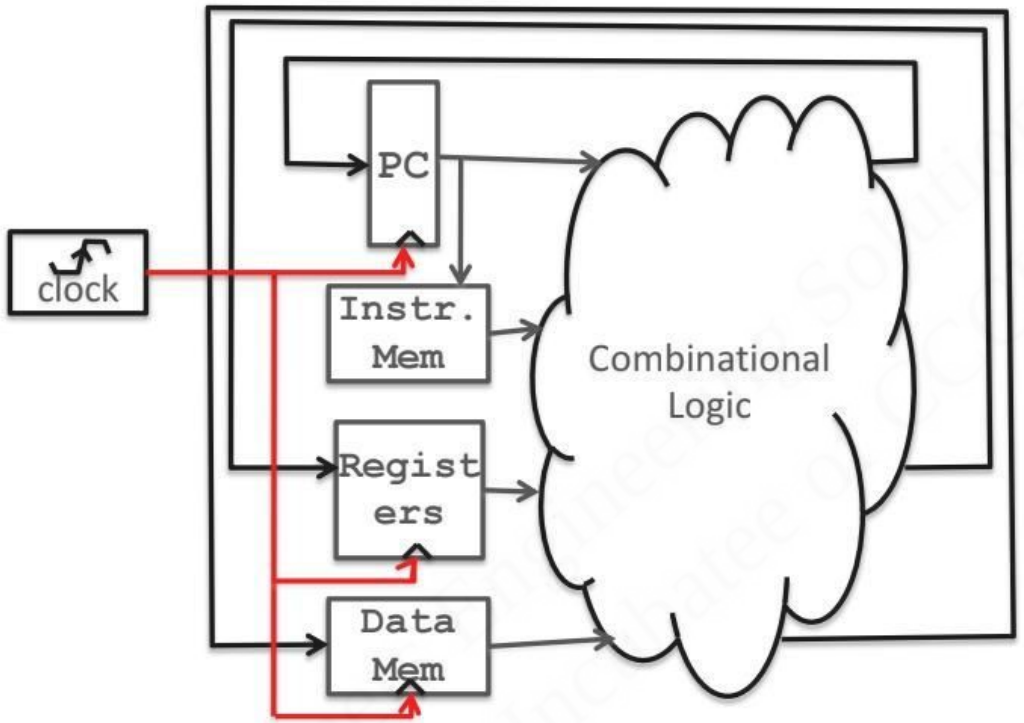
Components of a Computer



The CPU

- Processor (CPU): the active part of the computer that does all the work (data manipulation and decision-making)
- Datapath: portion of the processor that contains hardware necessary to perform operations required by the processor
- Control: portion of the processor (also in hardware) that tells the datapath what needs to be done

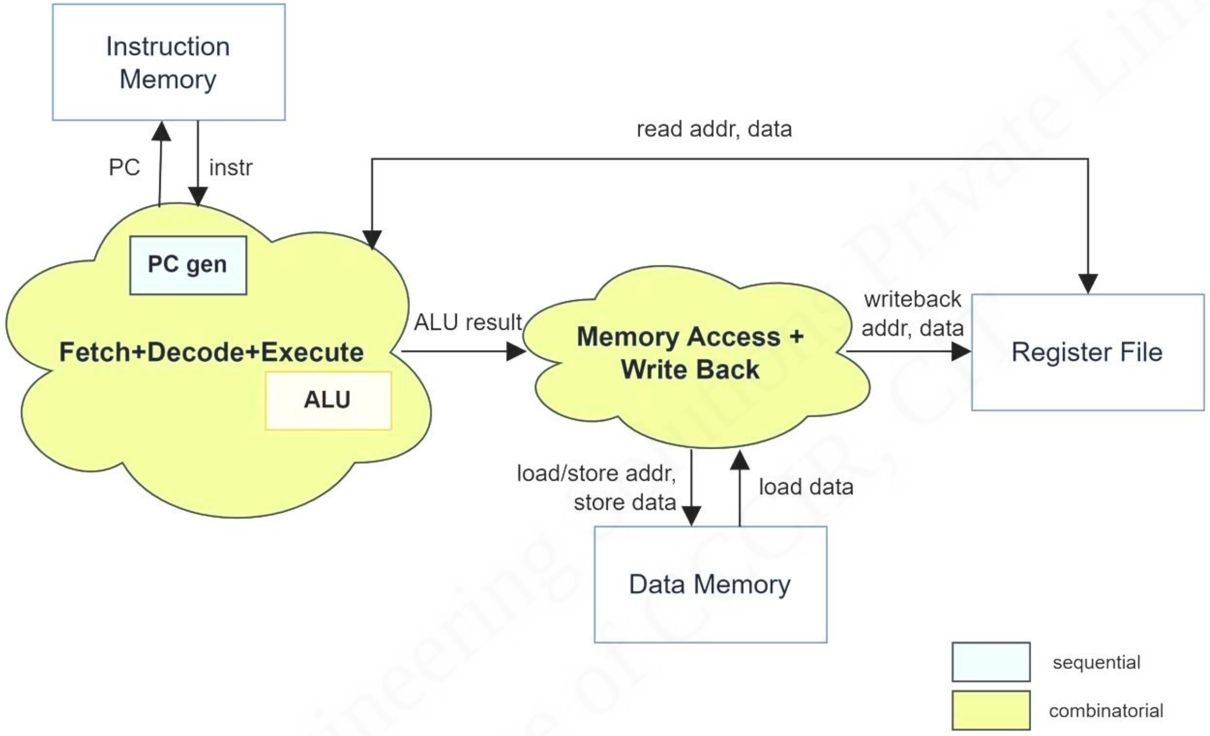
One-Instruction-Per-Cycle RISC-V Machine

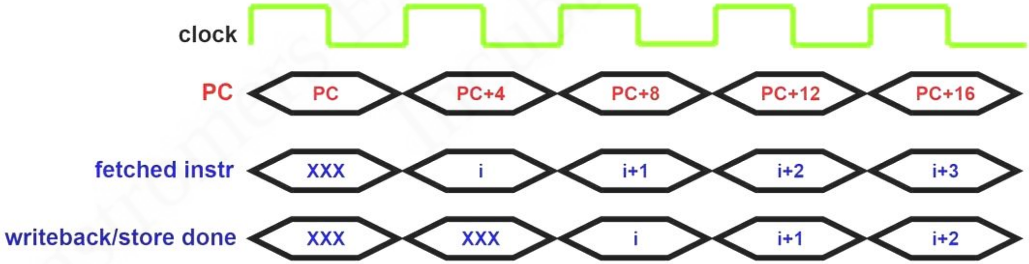


- One clock tick => one instruction
- Current state outputs => inputs to combinational logic => outputs settle at the values of state before next clock edge
- Rising clock edge:
  - all state elements are updated with combinational logic outputs
  - execution moves to next clock cycle

What is special about Instruction Memory?

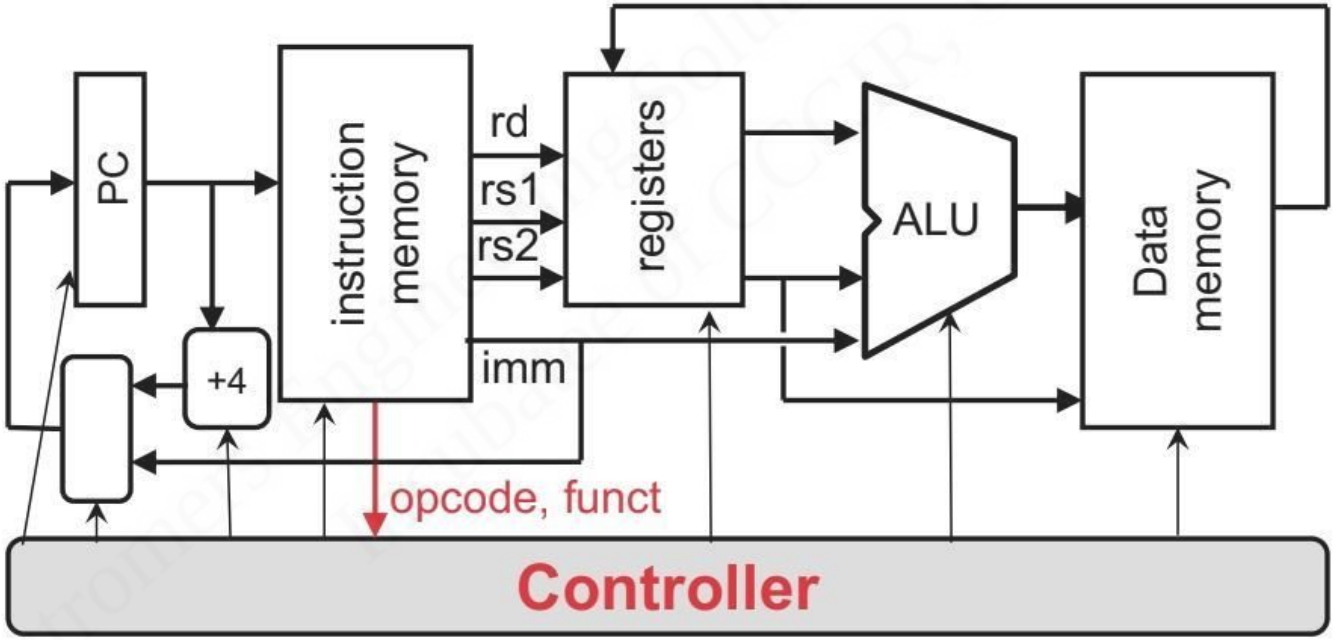
Why is Instruction Memory special?


Instruction Memory
PC
instr
read addr, data
PC gen
Fetch+Decode+Execute
ALU
ALU result
Memory Access + Write Back
writeback addr, data
Register File
load/store addr, store data
load data
Data Memory
sequential
combinatorial


clock
PC
PC
PC+4
PC+8
PC+12
PC+16
fetched instr
XXX
i
i+1
i+2
i+3
writeback/store done
XXX
XXX
i
i+1
i+2

Datapath and Control

- Datapath designed to support data transfers required by instructions
- Controller causes correct transfers to happen



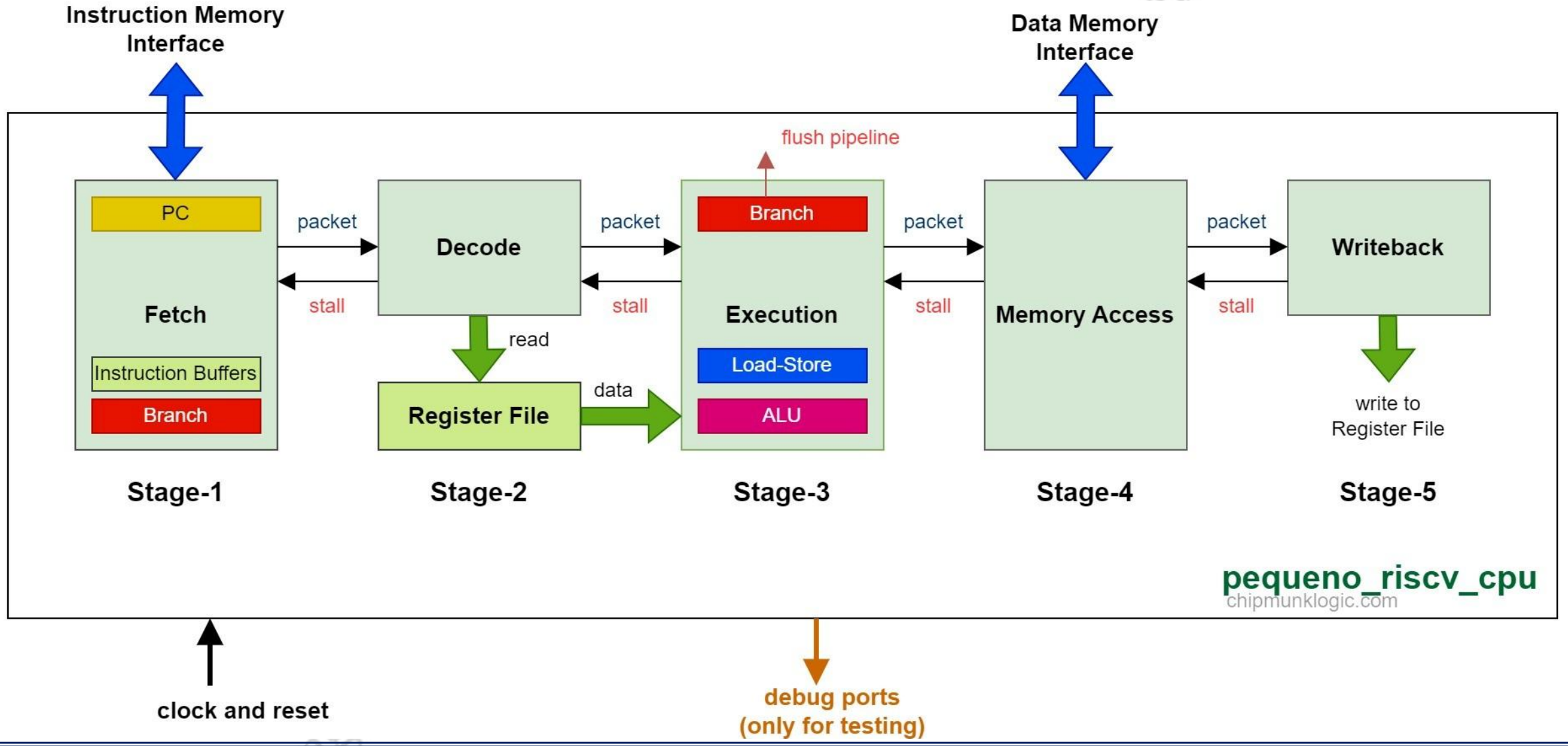
Stages of the Datapath : Overview

- Problem: a single, “monolithic” block that “executes an instruction” (performs all necessary operations beginning with fetching the instruction) would be too bulky and inefficient

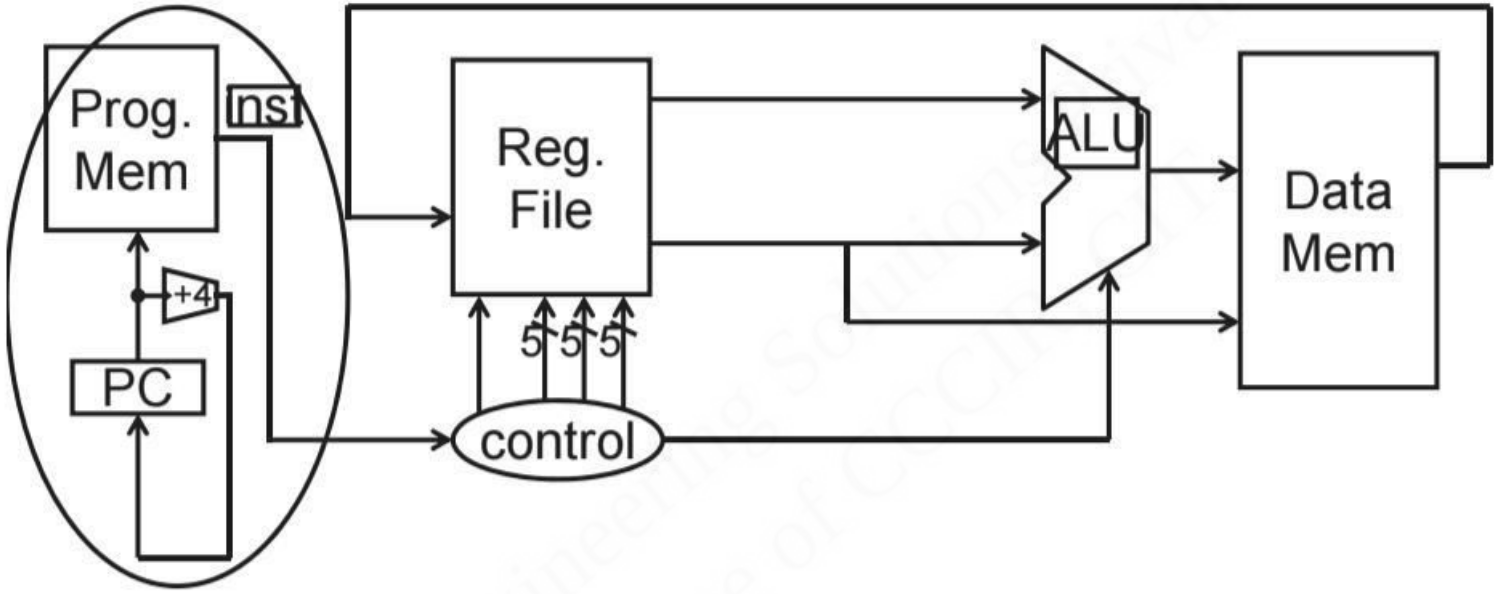
- Solution: break up the process of “executing an instruction” into stages, and then connect the stages to create the whole datapath
  - smaller stages are easier to design
  - easy to optimize (change) one stage without touching the others (modularity)

Five Stages of Instruction Execution

- Stage 1: Instruction Fetch (IF)
- Stage 2: Instruction Decode (ID)
- Stage 3: Execute (EX): ALU (Arithmetic-Logic Unit)
- Stage 4: Memory Access (MEM)
- Stage 5: Register Write (WB)



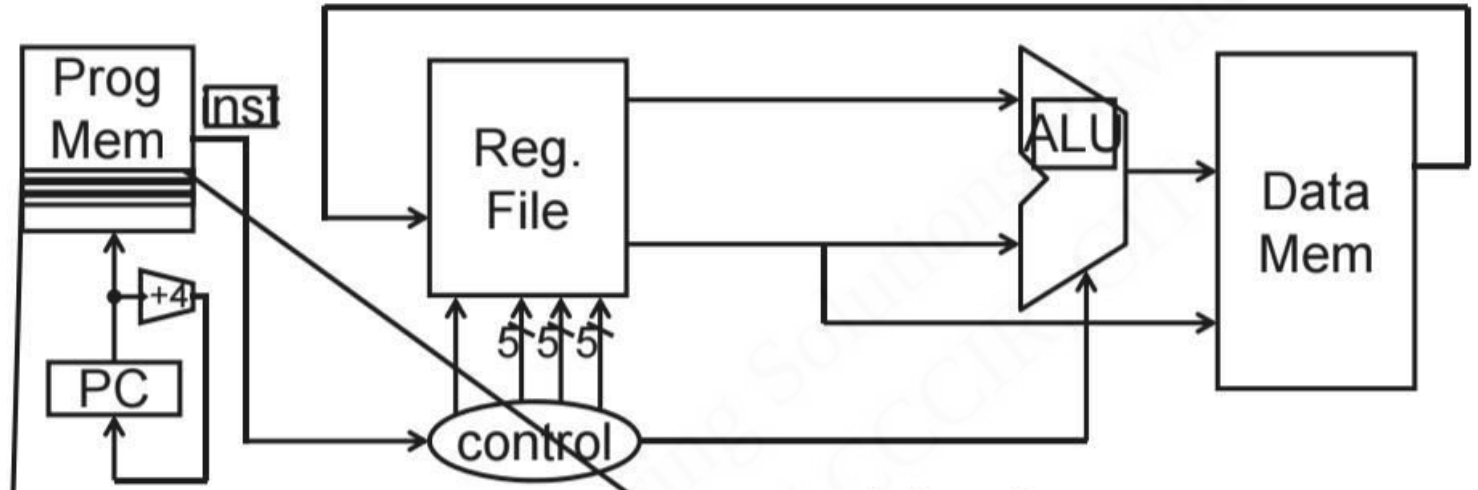
Stage 1: Instruction Fetch



Fetch 32-bit instruction from memory

Increment PC = PC + 4

Instruction Processing



Instructions:

stored in memory, encoded in binary

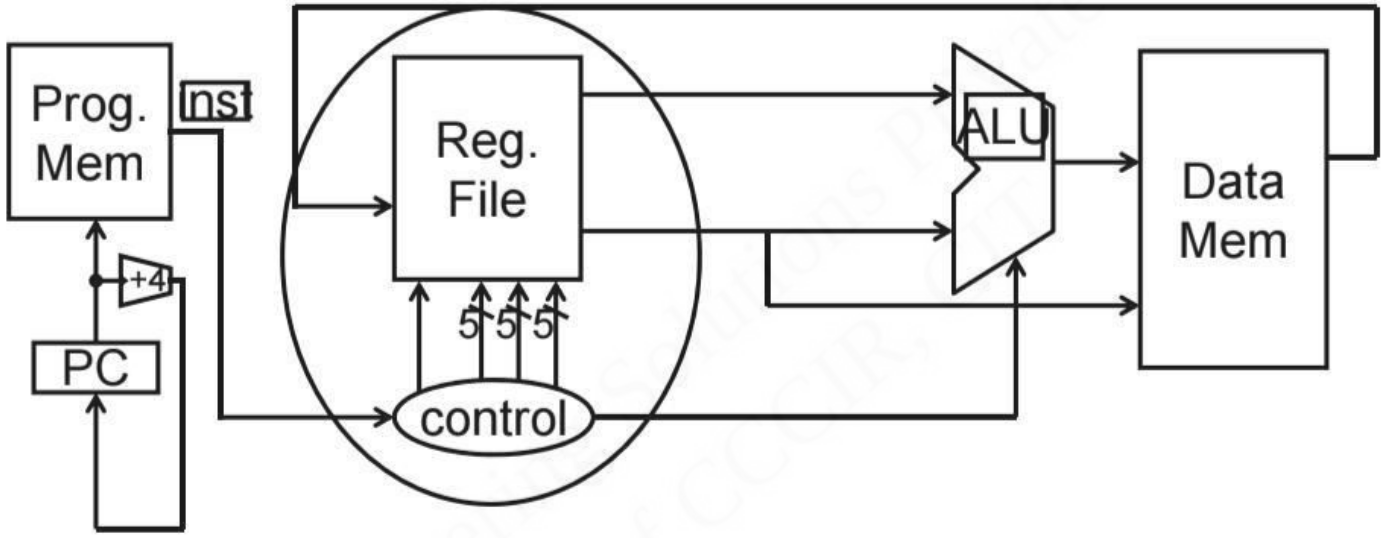
```
00100000000001000000000000001010
00100000000000100000000000000000
00000000001000100001100000101010
```

A basic processor

- fetches
- decodes
- executes

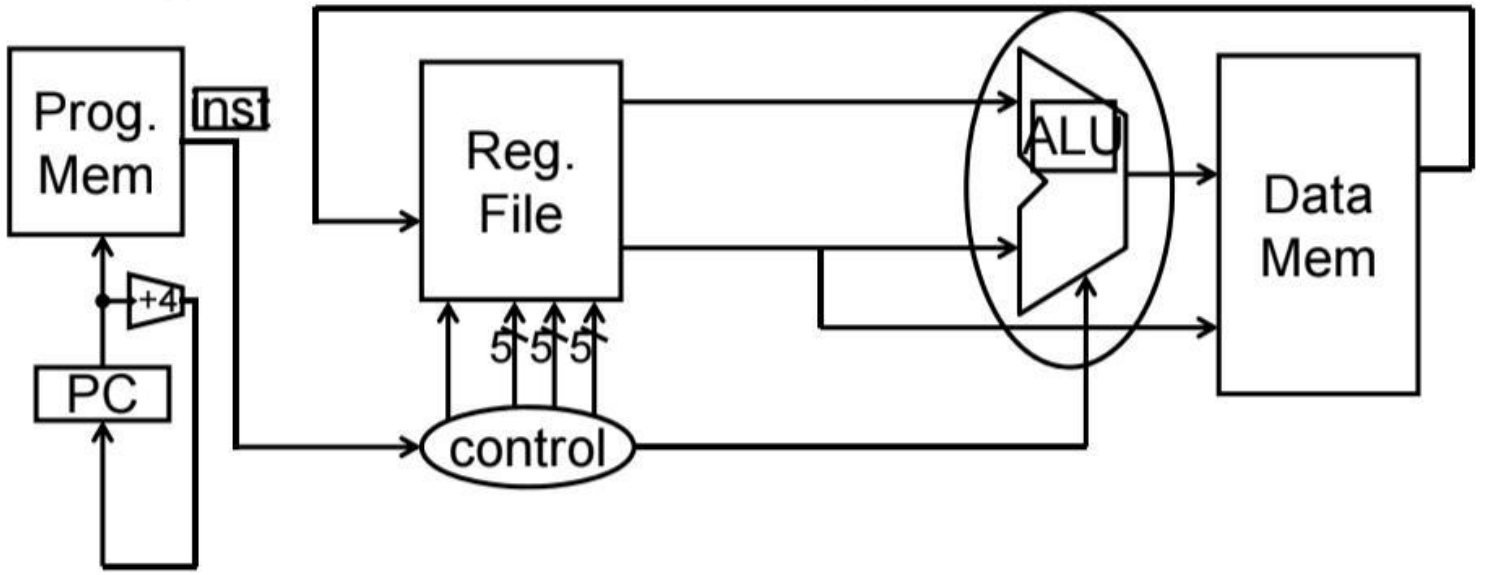
one instruction at a time

Stage 2: Instruction Decode



Gather data from the instruction
Read opcode; determine instruction type, field lengths
Read in data from register file
(0, 1, or 2 reads for `jump`, `addi`, or `add`, respectively)

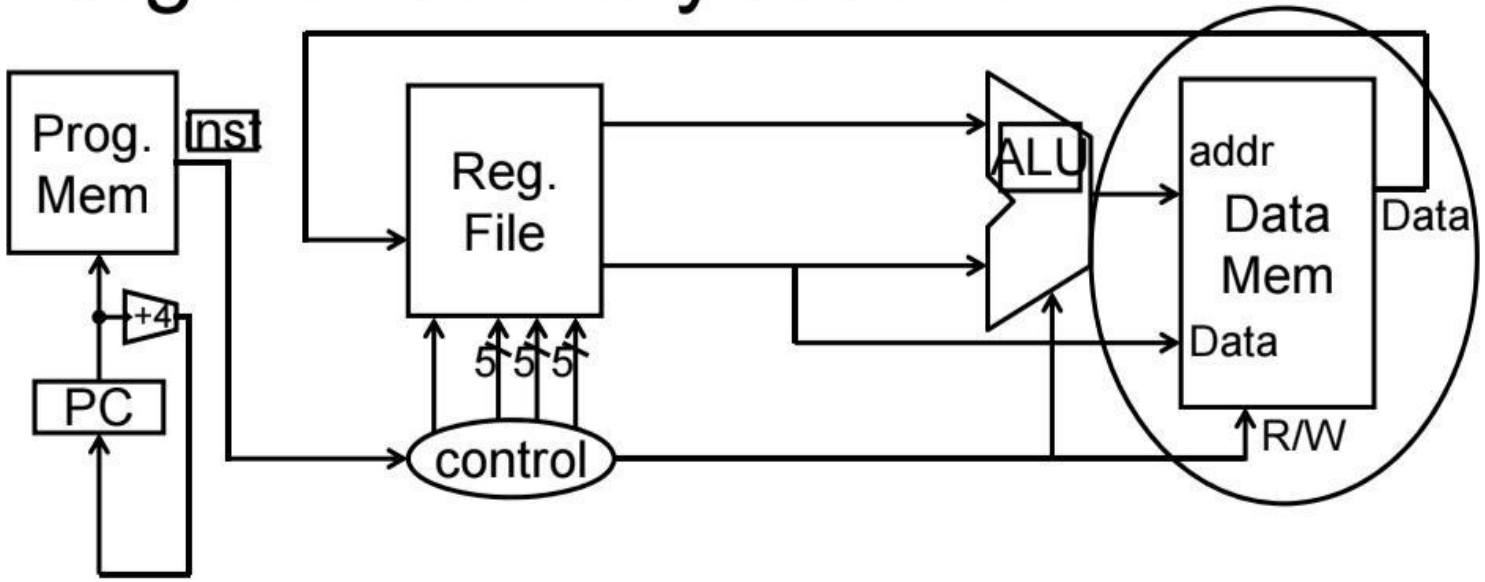
Stage 3: Execution (ALU)



Useful work done here (+, -, *, /), shift, logic operation, comparison (slt)

Load/Store? lw x2, x3, 32 → Compute address

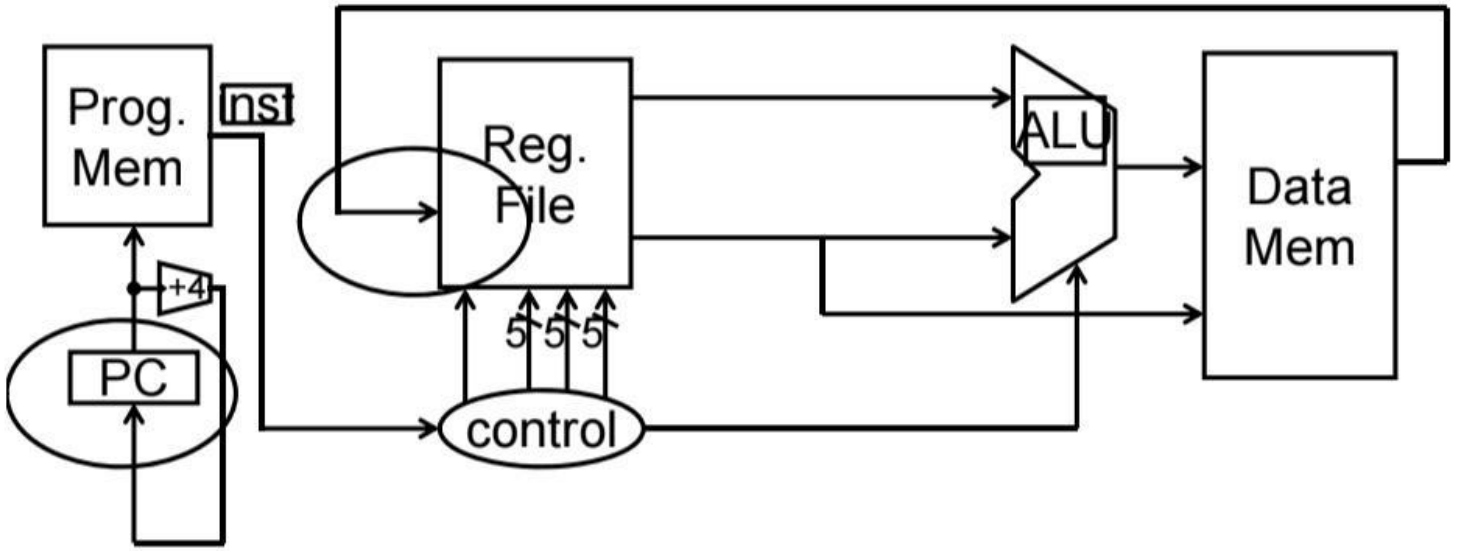
Stage 4: Memory Access

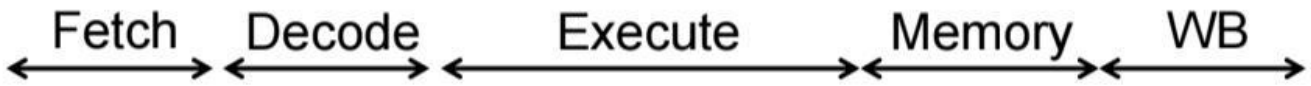


Used by load and store instructions only

Other instructions will skip this stage

Stage 5: Writeback





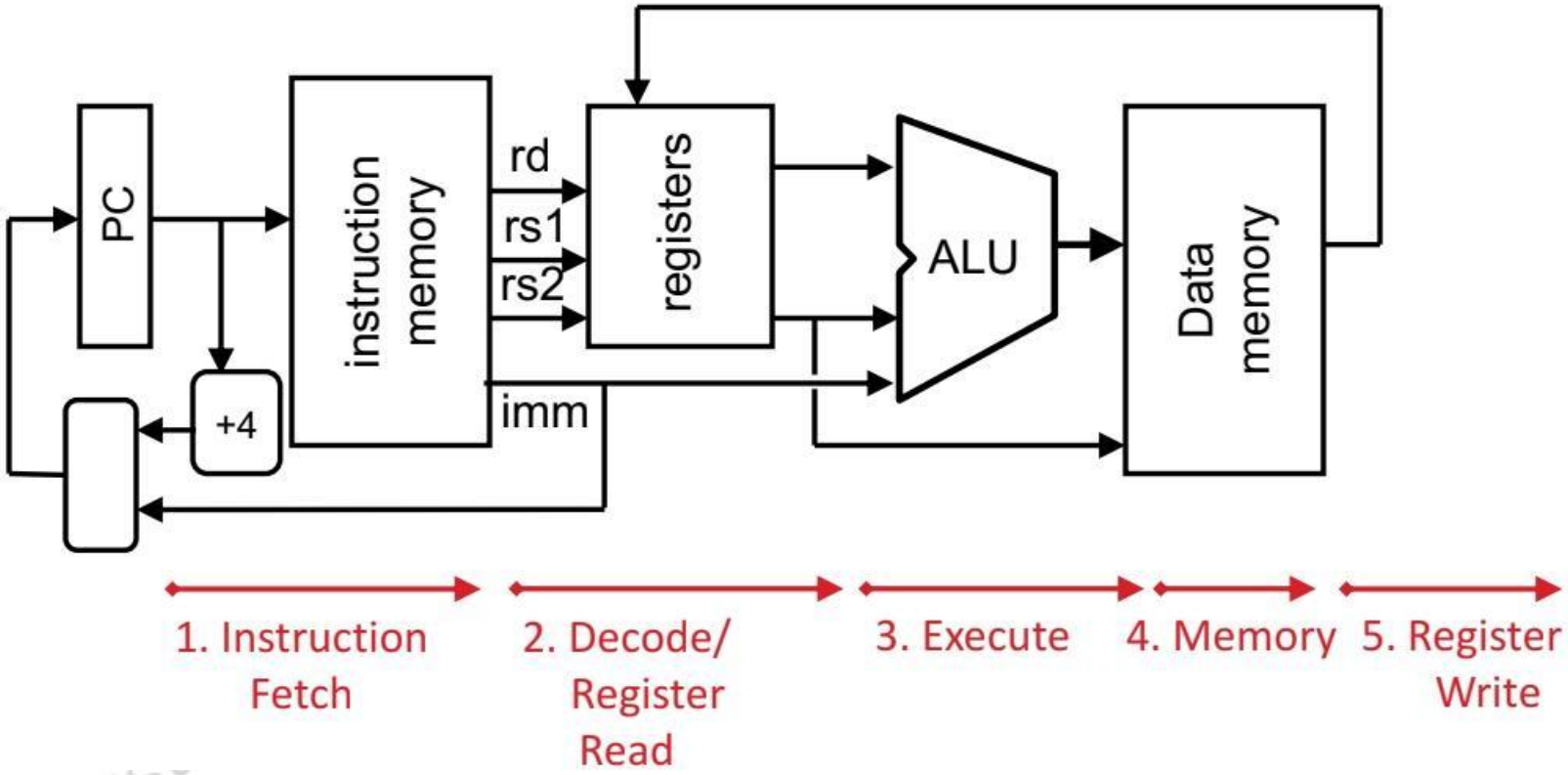
Write to register file

- For arithmetic ops, logic, shift, etc, load. What about stores?

Update PC

- For branches, jumps

Stages of Execution on Datapath



RISC-V Design Principles

Simplicity favors regularity

- 32 bit instructions

Smaller is faster

- Small register file

Make the common case fast

- Include support for constants

Good design demands good compromises

- Support for different type of interpretations/classes

RISC-V Instruction Types

- Arithmetic/Logical
  - R-type: result and two source registers, shift amount
  - I-type: result and source register, shift amount in 16-bit immediate with sign/zero extension
  - U-type: result register, 16-bit immediate with sign/zero extension
- Memory Access
  - I-type for loads and S-type for stores
  - load/store between registers and memory
  - word, half-word and byte operations
- Control flow
  - UJ-type: jump-and-link
  - I-type: jump-and-link register
  - SB-type: conditional branches: pc-relative addresses

RISC-V instruction formats

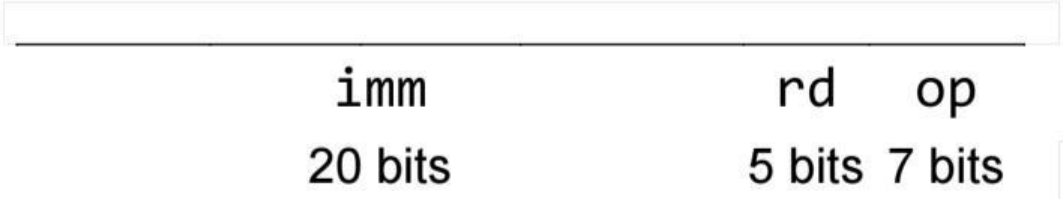
All RISC-V instructions are 32 bits long, have 4 formats

- R-type

- I-type

- S-type (SB-type)

- U-type (UJ-type)



Levels of Interpretation: Instructions

```
for (i = 0; i < 10; i++)
    printf("go cucs");
```

High Level Language
- C, Java, Python, ADA, …
- Loops, control flow, variables

```
main:  addi x2, x0, 10
       addi x1, x0, 0
loop:  slt x3, x1, x2
       ...
```

Assembly Language
- No symbols (except labels)
- One operation per statement
- "human readable machine language"

10 x2 x0 op=addi

```
00000000101000010000000010011
00100000000000100000000000010000
00000000001000100001100000101010
```

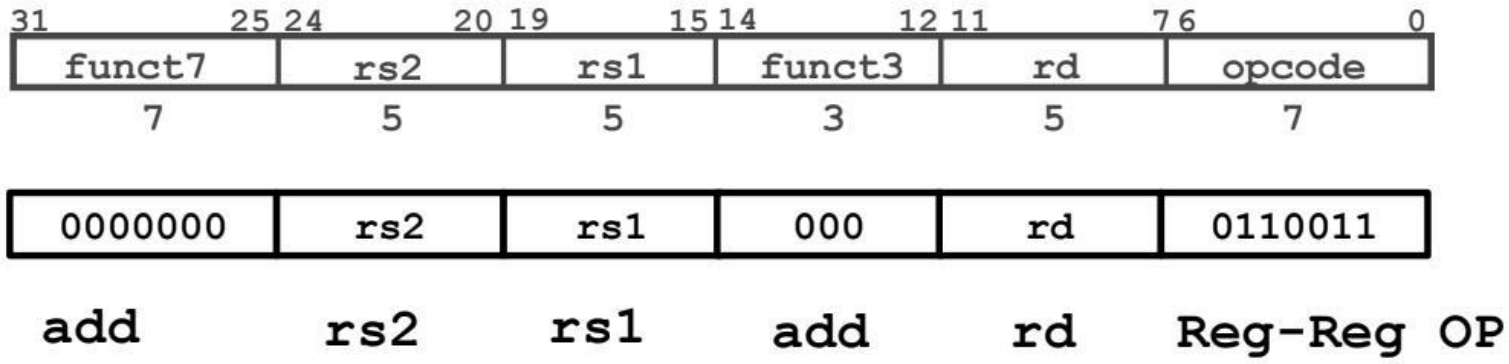
Machine Language
- Binary-encoded assembly
- Labels become addresses
- **The language of the CPU**

Instruction Set Architecture

ALU, Control, Register File, …

Machine Implementation
(Microarchitecture)

Implementing the **add** instruction



add rd, rs1, rs2

- Instruction makes two changes to machine's state:
  - **Reg[rd] = Reg[rs1] + Reg[rs2]**
  - **PC = PC + 4**

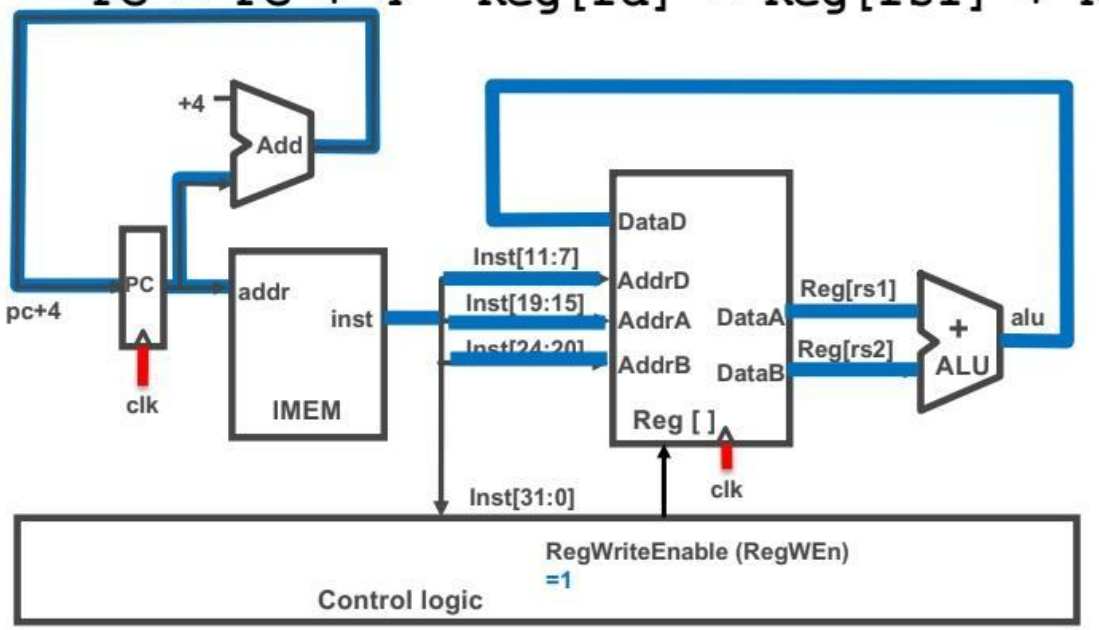
Implementing other R-Format instructions

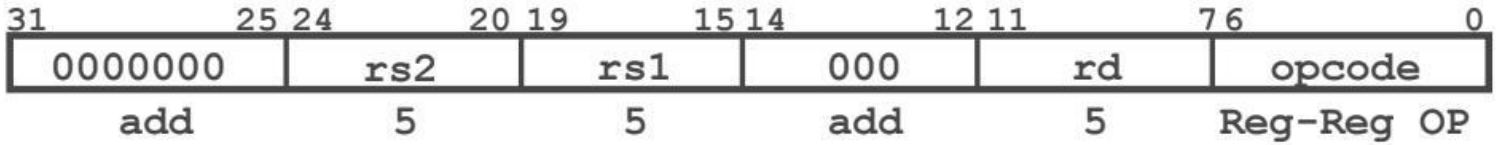
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

- All implemented by decoding funct3 and funct7 fields and selecting appropriate ALU function

Datapath for add

PC = PC + 4 Reg[rd] = Reg[rs1] + Reg[rs2]





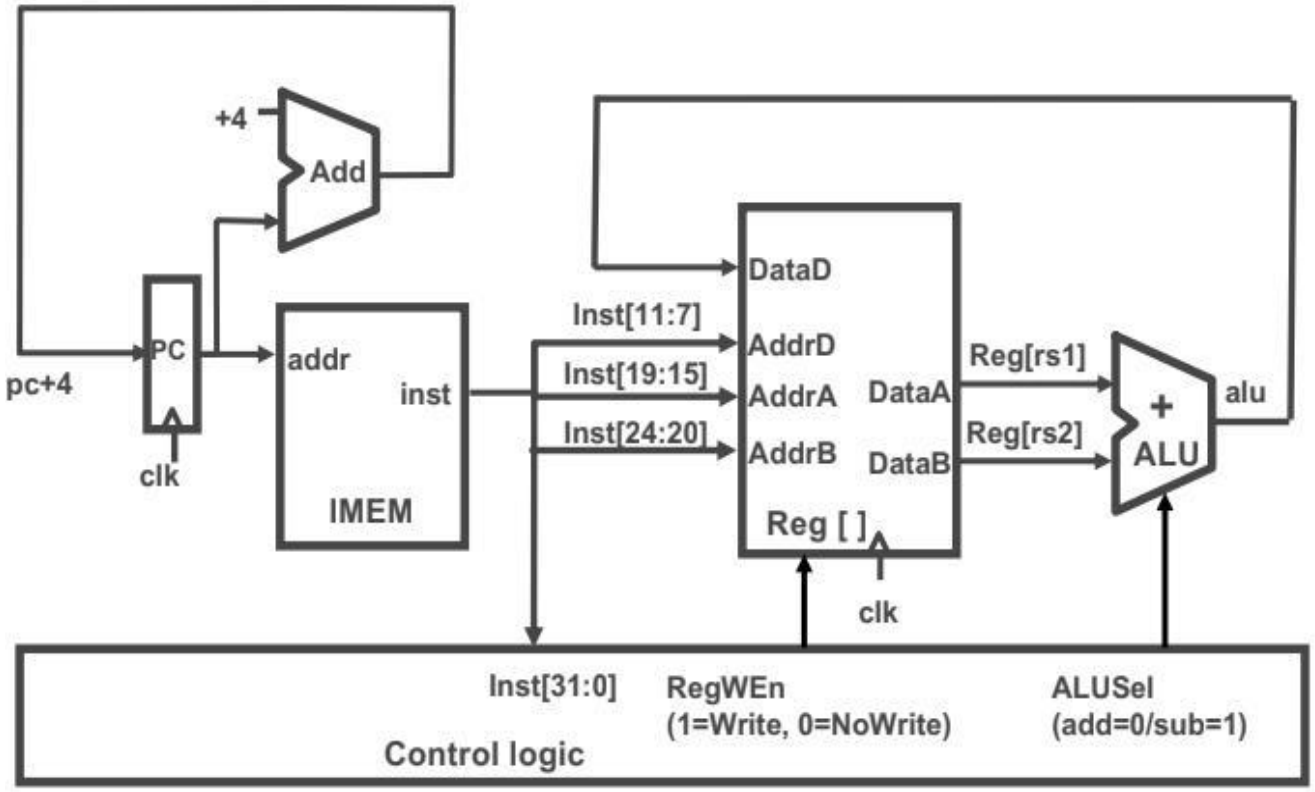
Implementing the **sub** instruction

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub

`sub rd, rs1, rs2`

- Almost the same as add, except now have to subtract operands instead of adding them
- **`inst[30]`** selects between add and subtract

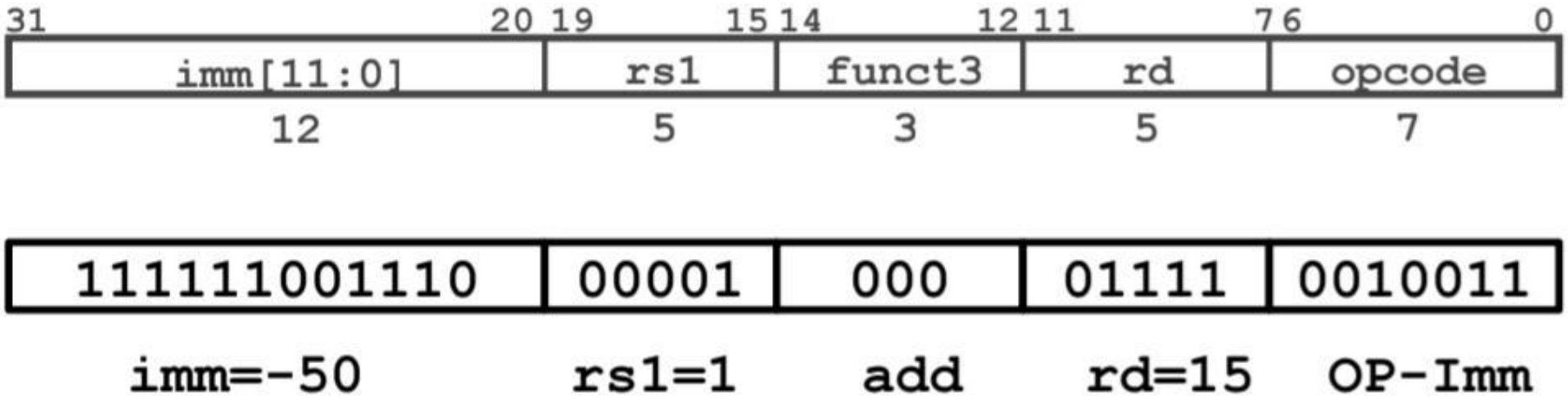
Datapath for add/sub



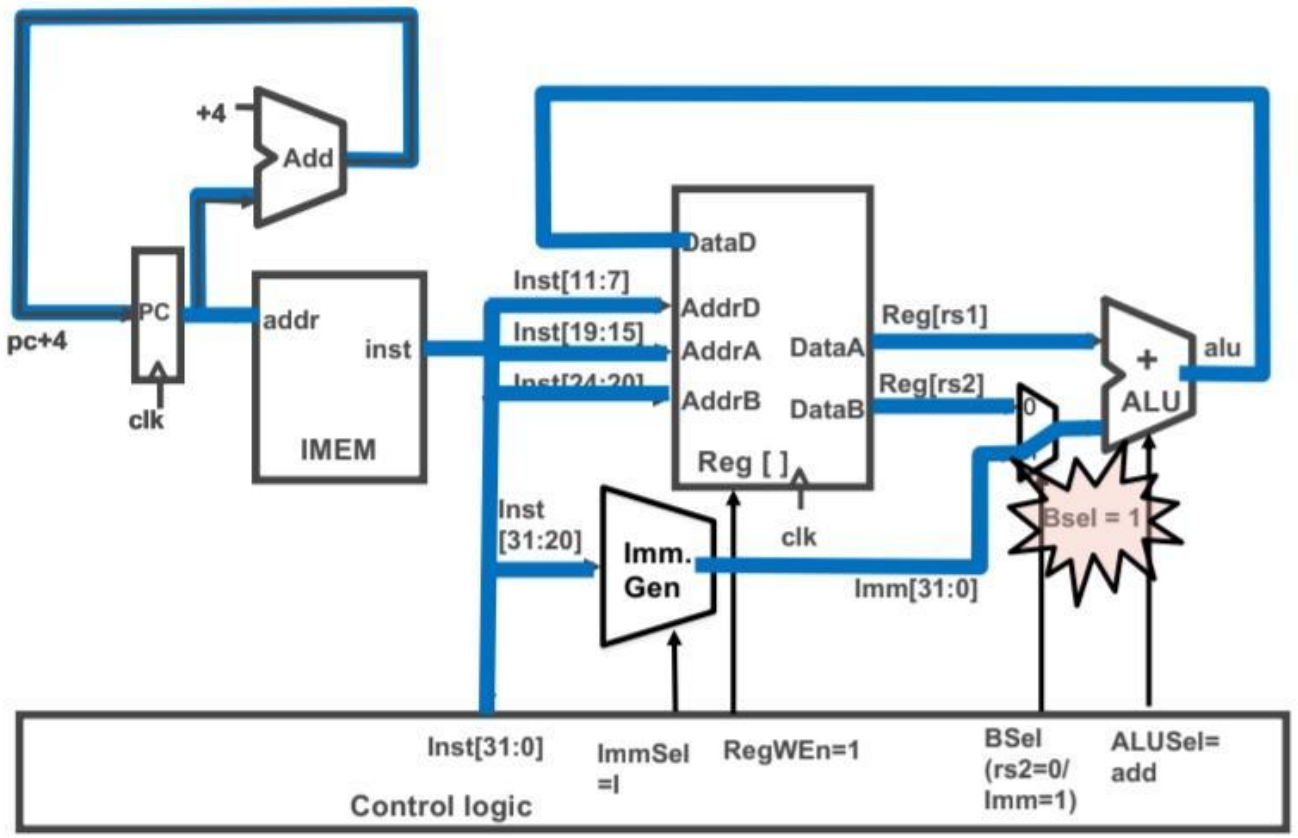
Implementing I-Format - **addi** instruction

- **RISC-V Assembly Instruction:**

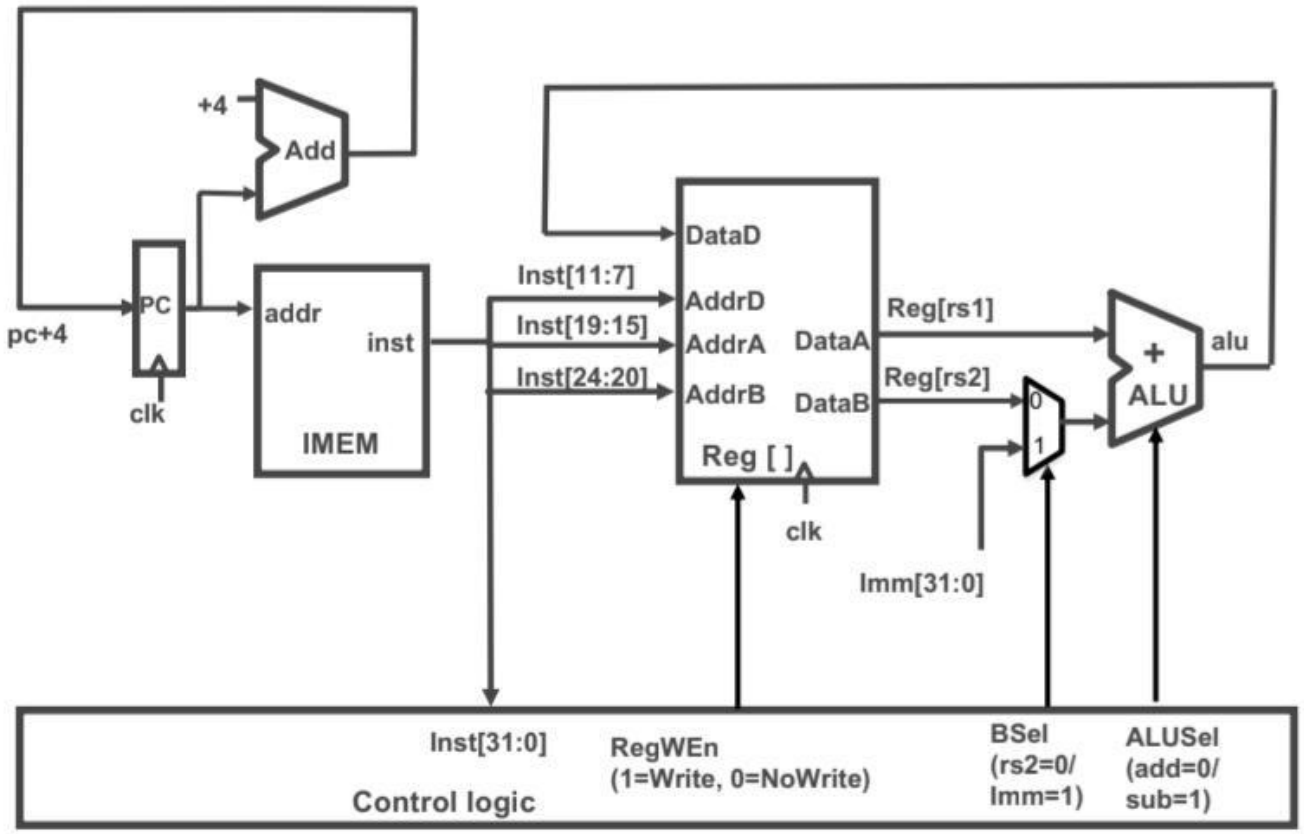
```
addi   x15,x1,-50
```



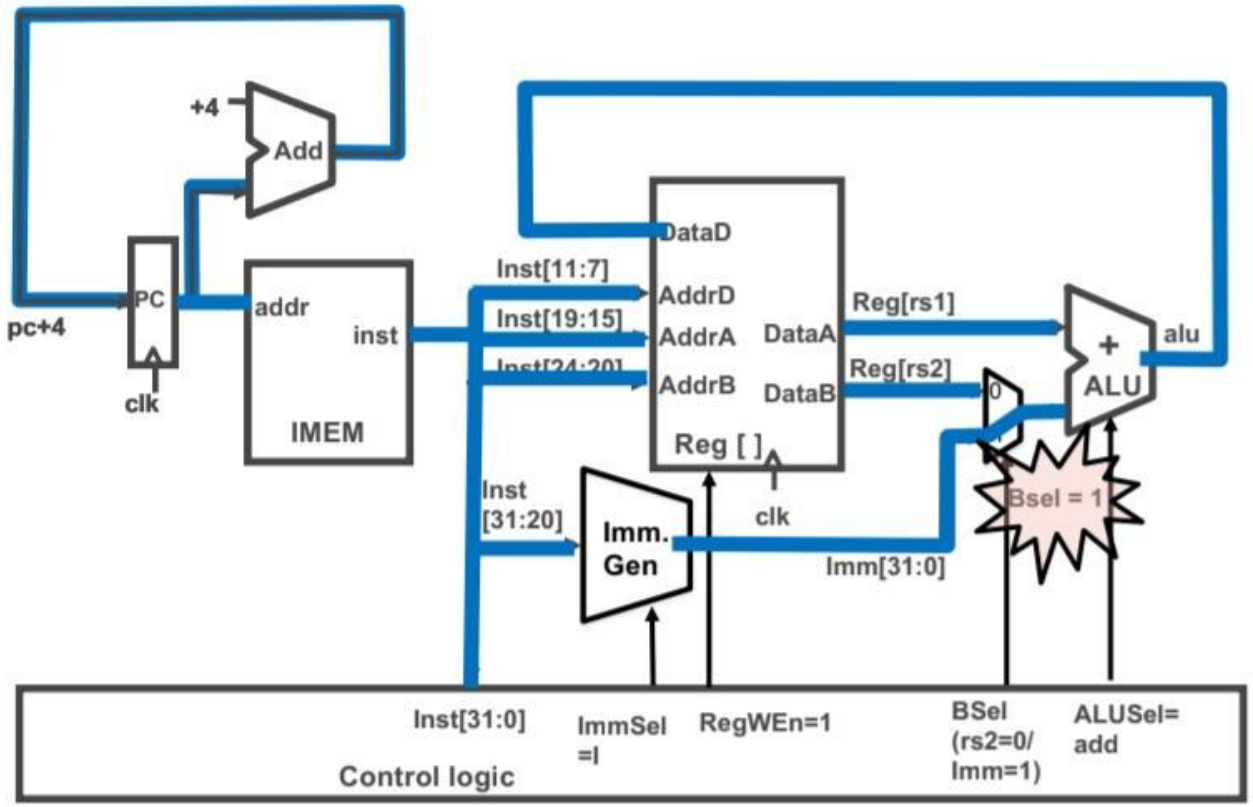
Adding addi to Datapath



Adding `addi` to Datapath



Adding `addi` to Datapath



Matrix Multiplication

```
#include <stdio.h>
#define N 3  // Matrix size (N x N)
void matrix_multiply(int A[N][N],
int B[N][N], int C[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            } } } }
int main() {
    int A[N][N] = {{1, 2, 3},
                   {4, 5, 6},
                   {7, 8, 9}};
    int B[N][N] = {{9, 8, 7},
                   {6, 5, 4},
                   {3, 2, 1}};
    int C[N][N] = {0};
    matrix_multiply(A, B, C);
    printf("Resultant Matrix C:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", C[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Simulation result using spike simulator

Compilation command: riscv64-unknown-elf-gcc main.c
Simulation command: Spike pk -s a.out

```
copi-001@copi001-OptiPlex-7050:~/spike/mat$ riscv64-unknown-elf-gcc main.c
copi-001@copi001-OptiPlex-7050:~/spike/mat$ spike pk -s a.out
bbl loader
Resultant Matrix C:
30 24 18
84 69 54
138 114 90
1750 ticks
84242 cycles
84242 instructions
1.00 CPI
```

Matrix Multiplication

```
#include <stdio.h>
unsigned long read_cycles(void)
{
  unsigned long cycles;
  asm volatile ("rdcycle %0" : "=r" (cycles));
  return cycles;
}
#define N 3  // Matrix size (N x N)
void matrix_multiply(int A[N][N],
int B[N][N], int C[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
                C[i][j] = 0;
                for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
                } } } }
int main() {
    int A[N][N] = {{1, 2, 3},
                   {4, 5, 6},
                   {7, 8, 9}};
unsigned long start, stop;
start = read_cycles();
int B[N][N] = {{9, 8, 7},
                {6, 5, 4},
                {3, 2, 1}};
    int C[N][N] = {0};
    matrix_multiply(A, B, C);
     stop = read_cycles();
     printf(" cycle :%ld\n", stop - start);
    printf("Resultant Matrix C:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
                printf("%d ", C[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Simulation result using spike simulator

Compilation command: riscv64-unknown-elf-gcc main.c
Simulation command: Spike pk a.out
Assembly generation command: riscv64-unknown-elf-gcc -S main.c
Assembly file view command: cat main.s

```
copi-001@copi001-OptiPlex-7050:~/spike/mat$ riscv64-unknown-elf-gcc main.c
copi-001@copi001-OptiPlex-7050:~/spike/mat$ spike pk a.out
bbl loader
Result matrix C:
 cycle :1910
Resultant Matrix C:
30 24 18
84 69 54
138 114 90
```

Matrix Transpose

```
#include <stdio.h>
#define N 4
 void transpose(int A[][N], int B[][N])
{
      int i, j;
      for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
            // Assigns the transpose of
element A[j][i] to
            // B[i][j]
            B[i][j] = A[j][i];
}
int main()
{
      int A[N][N] = { { 1, 1, 1, 1 },
                  { 2, 2, 2, 2 },
                  { 3, 3, 3, 3 },
                  { 4, 4, 4, 4 } };
      int B[N][N], i, j;
      transpose(A, B);
      printf("Result matrix is \n");
      for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++)
            printf("%d ", B[i][j]);
      printf("\n");
      }
      return 0;
}
```

Simulation result using spike simulator

Compilation command: riscv64-unknown-elf-gcc main.c
Simulation command: Spike pk -s a.out

```
copi-001@copi001-OptiPlex-7050:~/spike/transpose$ riscv64-unknown-elf-gcc main.c
copi-001@copi001-OptiPlex-7050:~/spike/transpose$ spike pk a.out
bbl loader
Result matrix is
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
```

Matrix Addition

```
#include <stdio.h>
#define ROWS 2
#define COLUMNS 3
void addMatrices(int
A[ROWS][COLUMNS], int
B[ROWS][COLUMNS], int
C[ROWS][COLUMNS]) {
      for (int i = 0; i < ROWS; i++) {
      for (int j = 0; j < COLUMNS; j++) {
            C[i][j] = A[i][j] + B[i][j];
      }
      }
}
int main() {
      int A[ROWS][COLUMNS] = {
      {1, 2, 3},
      {4, 5, 6}
      };
int B[ROWS][COLUMNS] = {
      {6, 5, 4},
      {3, 2, 1}
      };
      int C[ROWS][COLUMNS];
      addMatrices(A, B, C);
      printf("Resultant Matrix (A + B):\n");
      for (int i = 0; i < ROWS; i++) {
      for (int j = 0; j < COLUMNS; j++) {
            printf("%d ", C[i][j]);
      }
      printf("\n");
      }
      return 0;
}
```

Simulation result using spike simulator

Compilation command: riscv64-unknown-elf-gcc main.c
Simulation command: Spike pk -s a.out

```
copi-001@copi001-OptiPlex-7050:~/spike/madd$ riscv64-unknown-elf-gcc main.c
copi-001@copi001-OptiPlex-7050:~/spike/madd$ spike pk -s a.out
bbl loader
Resultant Matrix (A + B):
7 7 7
7 7 7
1650 ticks
75388 cycles
75388 instructions
1.00 CPI
```

Matrix Subtraction

```
#include <stdio.h>
#define ROWS 2
#define COLUMNS 3
void subtractMatrices(int
A[ROWS][COLUMNS], int
B[ROWS][COLUMNS], int
C[ROWS][COLUMNS]) {
      for (int i = 0; i < ROWS; i++) {
      for (int j = 0; j < COLUMNS; j++) {
            C[i][j] = A[i][j] - B[i][j];
      }
      }
}
int main() {
      int A[ROWS][COLUMNS] = {
      {10, 20, 30},
      {40, 50, 60}
      };
       int B[ROWS][COLUMNS] = {
      {1, 2, 3},
      {4, 5, 6}
      };
      int C[ROWS][COLUMNS];
      subtractMatrices(A, B, C);
      printf("Resultant Matrix (A - B):\n");
      for (int i = 0; i < ROWS; i++) {
      for (int j = 0; j < COLUMNS; j++) {
            printf("%d ", C[i][j]);
      }
      printf("\n");
      }
      return 0;
}
```

Simulation result using spike simulator

Compilation command: riscv64-unknown-elf-gcc main.c
Simulation command: Spike pk -s a.out

```
copi-001@copi001-OptiPlex-7050:~/spike/sadd$ riscv64-unknown-elf-gcc main.c
copi-001@copi001-OptiPlex-7050:~/spike/sadd$ spike pk -s a.out
bbl loader
Resultant Matrix (A - B):
9 18 27
36 45 54
1650 ticks
75577 cycles
75577 instructions
1.00 CPI
```

Matrix Division

```
#include <stdio.h>
#define ROWS 2
#define COLUMNS 3
void divideMatrices(int
A[ROWS][COLUMNS], int
B[ROWS][COLUMNS], float
C[ROWS][COLUMNS]) {
      for (int i = 0; i < ROWS; i++) {
      for (int j = 0; j < COLUMNS; j++) {
            if (B[i][j] != 0)
            C[i][j] = (float)A[i][j] / B[i][j];
            else
            C[i][j] = 0.0; // or handle error
      } } }
int main() {
      int A[ROWS][COLUMNS] = {
      {10, 20, 30},
      {40, 50, 60}
      };
      int B[ROWS][COLUMNS] = {
      {2, 4, 5},
      {8, 10, 15}
      };
      float C[ROWS][COLUMNS];
      divideMatrices(A, B, C);
      printf("Resultant Matrix (A / B):\n");
      for (int i = 0; i < ROWS; i++) {
      for (int j = 0; j < COLUMNS; j++) {
            printf("%.2f ", C[i][j]);
      }
      printf("\n");
      }
      return 0;
}
```

Simulation result using spike simulator

Compilation command: riscv64-unknown-elf-gcc main.c
Simulation command: Spike pk -s a.out

```
copi-001@copi001-OptiPlex-7050:~/spike/mdiv$ riscv64-unknown-elf-gcc main.c
copi-001@copi001-OptiPlex-7050:~/spike/mdiv$ spike pk -s a.out
bbl loader
Resultant Matrix (A / B):
5.00 5.00 6.00
5.00 5.00 4.00
2000 ticks
99347 cycles
99347 instructions
1.00 CPI
```

Pseudo code for RISC-V Inline Assembly addition Code

```
“la x10,a"
"la x11, b"
"lw x5, 0(x10)"
"lw x6, 0(x11)"
"add x7, x5, x6"
"la x8, sum"
"sw x7, 0(x8)"
```

RISC-V Inline Assembly addition Code

```
#include <stdio.h>
void add_asm(void);
static int a=5;
static int b=6;
static int sum;

int main()
{

add_asm();
printf("sum = %d\n",sum);
}
void add_asm()
{
asm volatile (
"la x10,a\n\t"
"la x11, b\n\t"
"lw x5, 0(x10)\n\t"
"lw x6, 0(x11)\n\t"
"add x7, x5, x6\n\t"
"la x8, sum\n\t"
"sw x7, 0(x8)\n\t"
);
}
```

Pseudo code for RISC-V Inline Assembly code to swap two number

```
"la x10, a"
"lw x11, 0(x10)"
"la x12, b"
"lw x13, 0(x12)"
"sw x13, 0(x10)"
"sw x11, 0(x12)"
```

RISC-V Inline Assembly addition Code

```
#include <stdio.h>
void add_asm(void);
static int a=5;
static int b=6;
static int sum;

int main()
{

add_asm();
printf("sum = %d\n",sum);
}
void add_asm()
{
asm volatile (
"la x10,a\n\t"
"la x11, b\n\t"
"lw x5, 0(x10)\n\t"
"lw x6, 0(x11)\n\t"
"add x7, x5, x6\n\t"
"la x8, sum\n\t"
"sw x7, 0(x8)\n\t"
);
}
```

Thank you