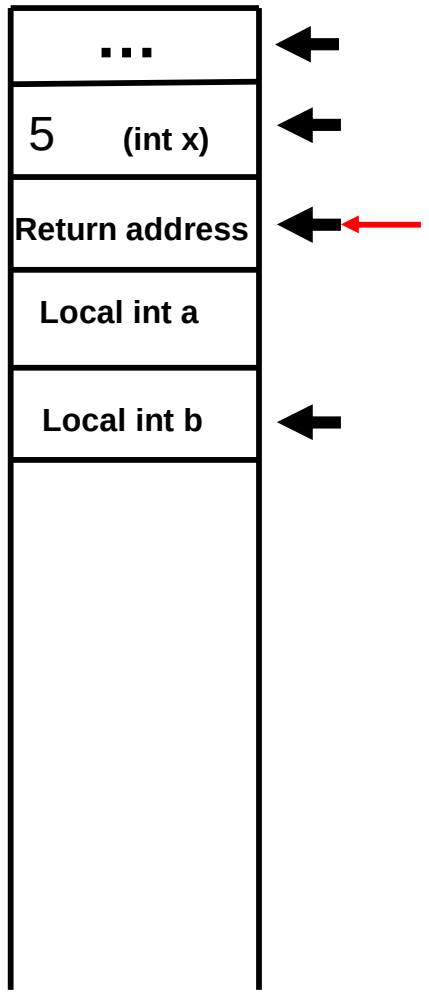
MIPS 1 Function Call

```
void B (int x) {
   int a, b;
     …

     return();
}
```

```
void A() {
     …
     B(5);
     …
}
```

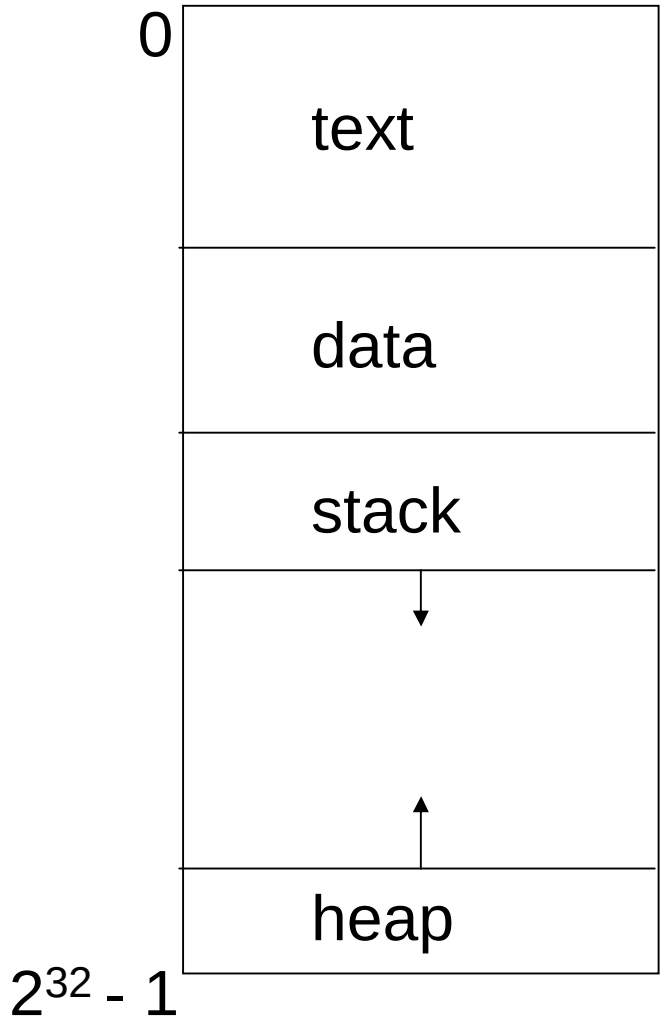
```
B:   ADDI R29, R29, 4
     SW 0(R29), R31
     ADDI R29, R29, 8
     …
     SUBI R29, R29, 16
     LW R31, 8(R29)
     JR R31


     ADDI R1, R0, 5
     ADDI R29, R29, 4
     SW 0(R29), R1
     JAL B
```

Function Call/Return Stack



Use of Main Memory by a Program

- Instructions (code, text)
- Data
  - Statically allocated
  - Stack allocated
  - Heap allocated



MIPS 1 Instruction Set

1. Conditional branch instructions
2. Floating point

MIPS 1 Branch Instructions

	Mnemonics	Example	Meaning
Conditional Branch	BEQ, BNE, BGEZ, BLEZ, BLTZ, BGTZ	BLTZ R2, -16	If R2 < 0, PC ← PC + 4 -16

= = != >= 0 <= 0 < 0 > 0

What about a condition like R1 >= R2

You could use the BGEZ instruction

Idea: Rewrite the condition as (R1 – R2) >= 0

```
SUB   R3,  R1,  R2      / R3 ← R1  -  R2
BGEZ  R3, target        / if R3 >= 0 goto target
```

Problem: Possibility of overflow

MIPS 1 Compare Instructions

	Mnemonics	Example	Meaning
Compare	SLT, SLTU, SLTI, SLTIU	SLT R1, R2, R3	R1← 1 if R2 < R3 ← 0 otherwise

S: Set, LT: If less than; I: Immediate; U: Unsigned

if (R1 >= R2) thenpart;

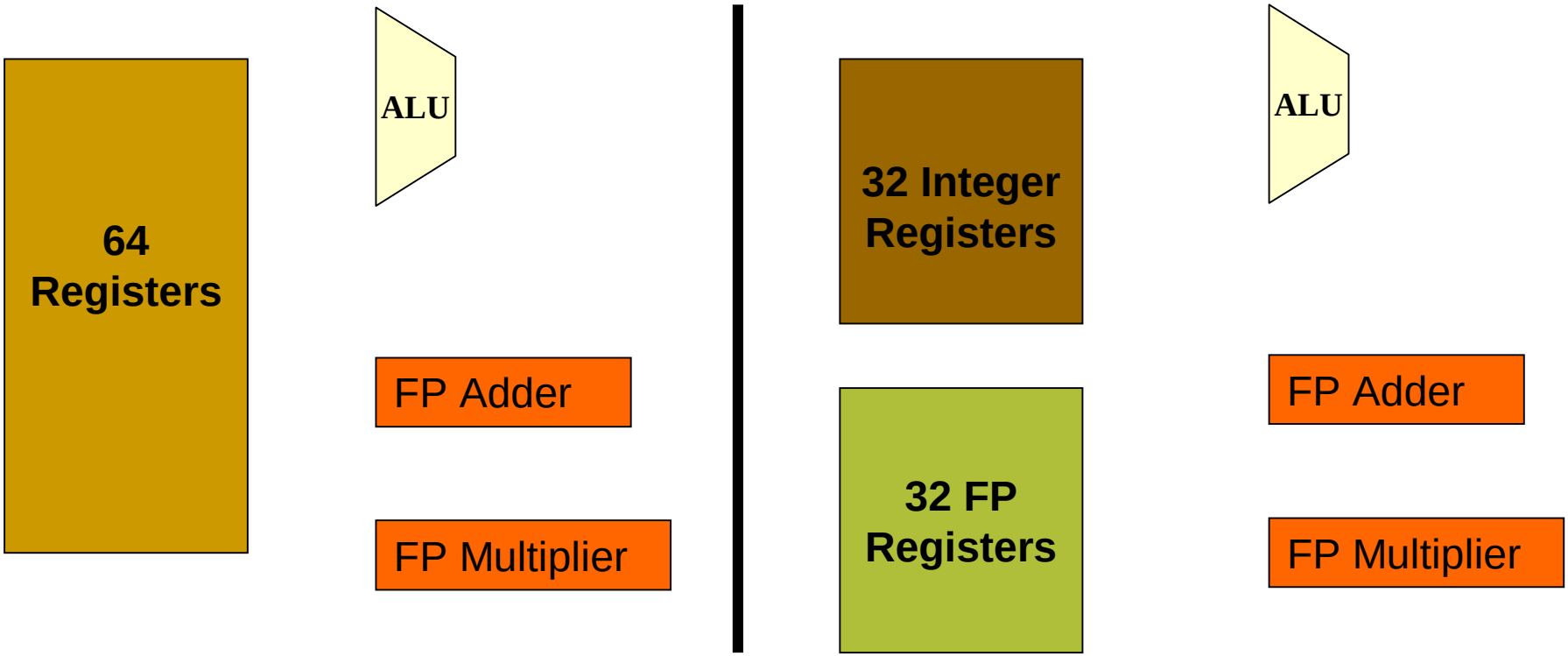
else elsepart

```
SLT   R3,  R1,  R2      / R3 ← 1  if  R1  <  R2
                        / if  R3 = = 0  goto  thenpart
BEQ   R3,  R0,  thenpart
```

MIPS 1 Floating Point

- Assume that there is a separate floating point register file
  - 32 32b floating point registers F0-F31
  - A double (64b floating point value) occupies 2 registers
    - Even-odd pair, such as F0,F1
    - Addressed as F0
- Additional instructions
  - Loads: LF (load float), LD (load double)
  - Arithmetic: ADDF (add float), ADDD (add double)

Rationale for Separate FP Register File?

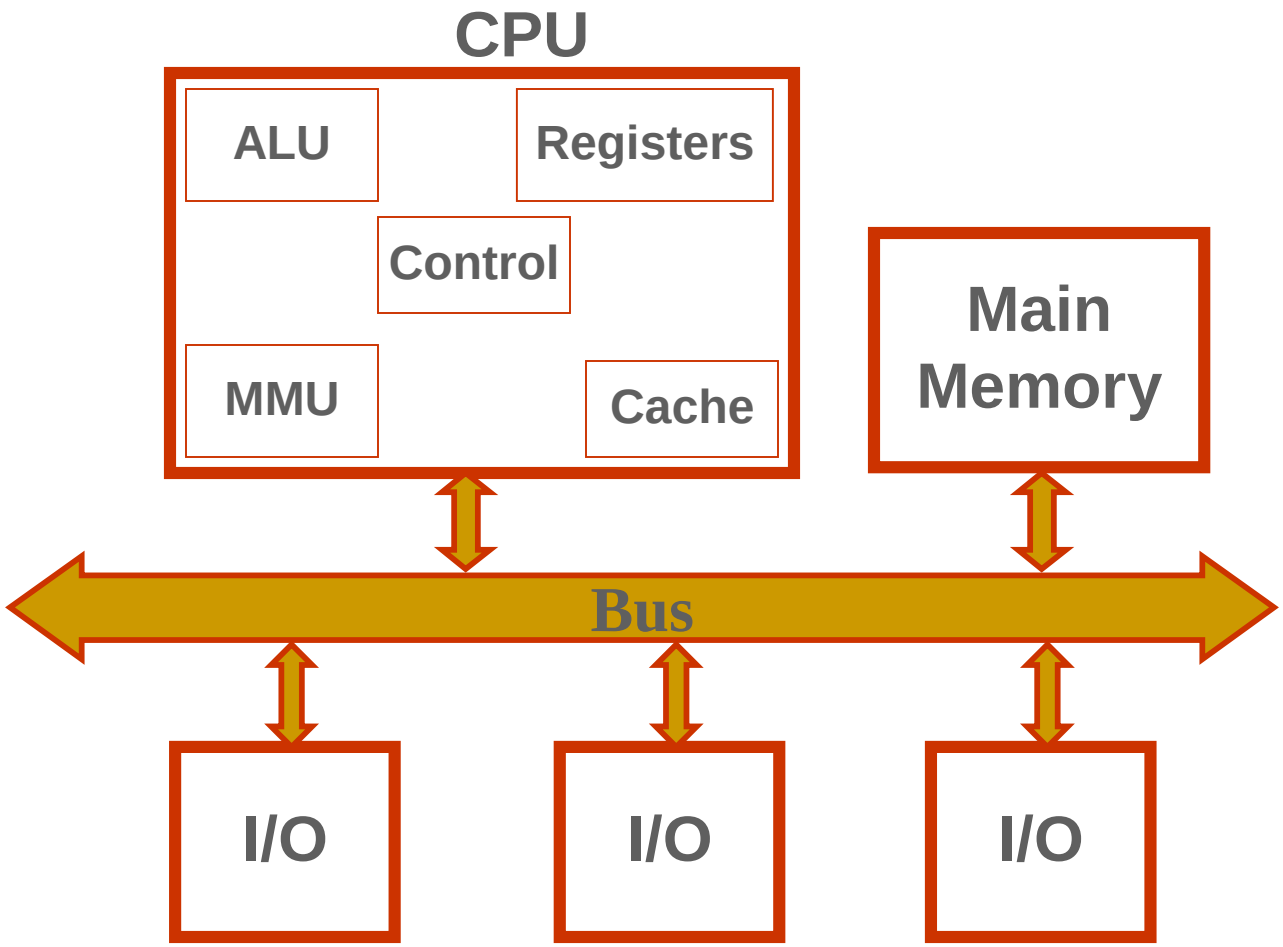


MIPS 1 Floating Point Code Example

```
double A[1024], B[1024];
for (i=0; i<1024; i++) A[i] = A[i] + B[i];
```

```
Loop:  LD      F0,  0(R1)
       LD      F2,  0(R2)
       ADDD    F4,  F0,  F2
       SD      0(R1),  F4
       ADDI    R1,  R1,  8
       ADDI    R2,  R2,  8
       BNE     R1,  R3,   Loop
```

Basic Computer Organization



Steps in Instruction Processing

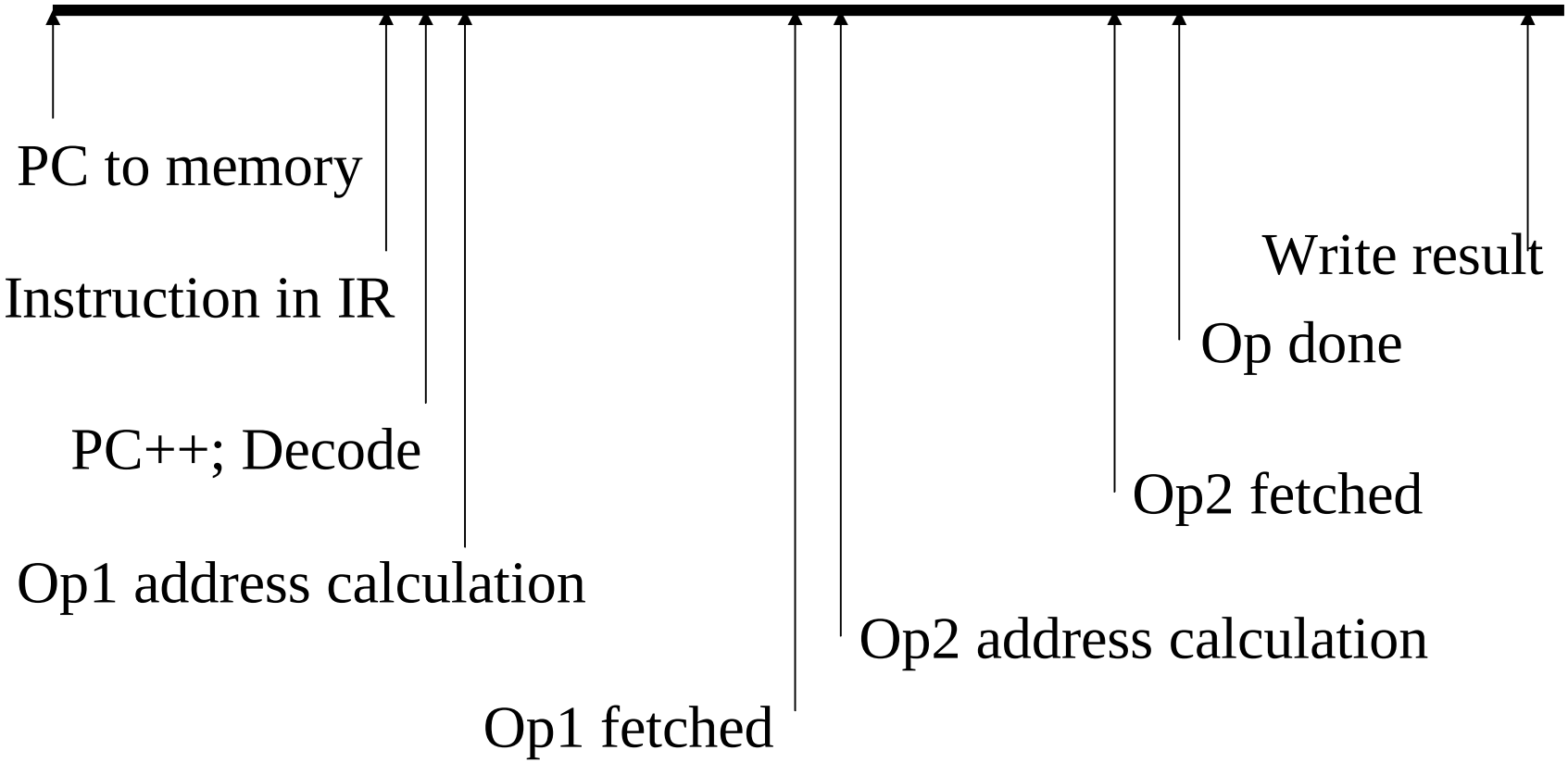
1. Fetch instruction from Main Memory to CPU
   - Get instruction whose address is in PC from memory into IR
   - Increment PC
2. Decode the instruction
   - Understand instruction, addressing modes, etc
   - Calculate memory addresses and fetch operands
3. Execute the required operation
   - Do the required operation
4. Write the result of the instruction

Steps in Instruction Execution

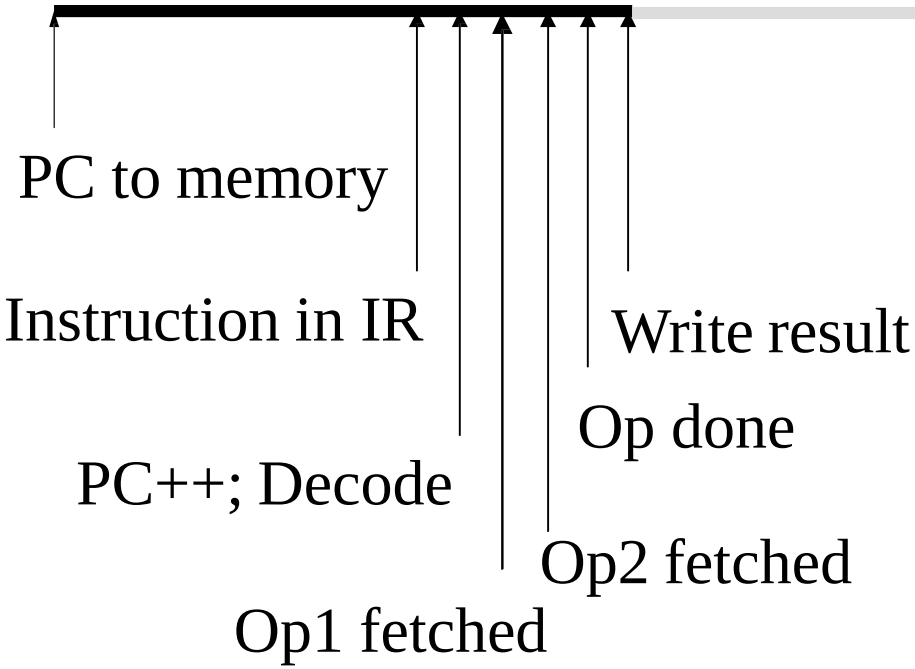
- Fetch instruction from memory to processor
  - IR = Memory[PC]; Increment PC
- Decode instruction and get its operands
  - Decode; Operands from registers/memory to ALU
- Execute the operation
  - Trigger appropriate functional hardware
  - If load/store, send access request to memory
- Write back the result
  - To destination register/memory

Timeline of events (CISC)

Processor/Memory Speed disparity ~2 orders of magnitude



Timeline of events (RISC)



Aside: CISC vs RISC Instructions

A[i++] = A[i] + B[i];

B[i] = B[i--] - 1;

RISC Code:

```
LW    R1,  0(R3)
LW    R2,  0(R4)
ADD   R5,  R1,  R2
SUBI  R2,  R2,  1
SW    0(R3), R5
SW    0(R4), R2
ADDI  R3,  R3,  4
SUBI  R4,  R4,  4
```

CISC Code:

```
add   (R3)+, (R3), (R4)
sub   (R4) -, (R4),  1
```

Instructions	Memory Accesses
RISC - 8	RISC - 4
CISC - 2	CISC - 5

We will assume that …

1. Activity is overlapped in time where possible
   - PC increment and instruction fetch from memory?
   - Instruction decode and effective address calculation
2. Load-store ISA: the only instructions that take operands from memory are loads & stores
3. Main memory delays are not typically seen by the processor
   - Otherwise the timeline is dominated by them
   - There is some hardware mechanism through which most memory access requests can be satisfied at processor speeds (cache memory)