CENG 3420
Computer Organization & Design
Lecture 05: Control Instructions

Textbook: Chapter 2.6-2.7

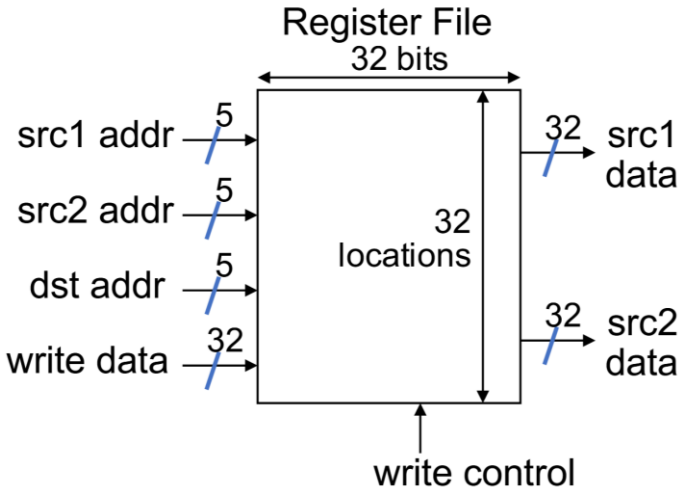
Zhengrong Wang

CSE Department, CUHK

zhengrongwang@cuhk.edu.hk

RISC-V Register File

- Registers are
  - Faster than main memory.
    - But larger register files are slower.
    - E.g., a 64-word file may be 50% slower than a 32-word file.
    - Read/write port increase impacts speed quadratically.
  - Easier for compiler to use.
    - (A*B)-(C*D)-(E*F) can do multiplies in any order vs. stack.
  - Can hold variables so that code density improves.
    - 5-bit to encode a register vs. 32-bit for a memory address.



- 32 32-bit general purpose registers.
- 2 read ports.
- 1 write ports.

RV32I Unprivileged Integer Register

- Return address ra/x1
  - Before function calls, explicitly save the return address in ra (usually pc + 4).
- Stack pointer sp/x2
  - Points to the logical top of the stack (which is to the lowest memory address, since the stack grows downward).
- Global pointer gp/x3
  - Holds the base address of the memory region containing global variables.
- Function arguments a0-a7/x10-x17
  - Holds the arguments of a function call.
  - Extra arguments (>8) are pushed into the stack.

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	—
x3	gp	Global pointer	—
x4	tp	Thread pointer	Callee
x5	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP args/return values	Caller
f12-17	fa2-7	FP args	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

Instruction Encoding

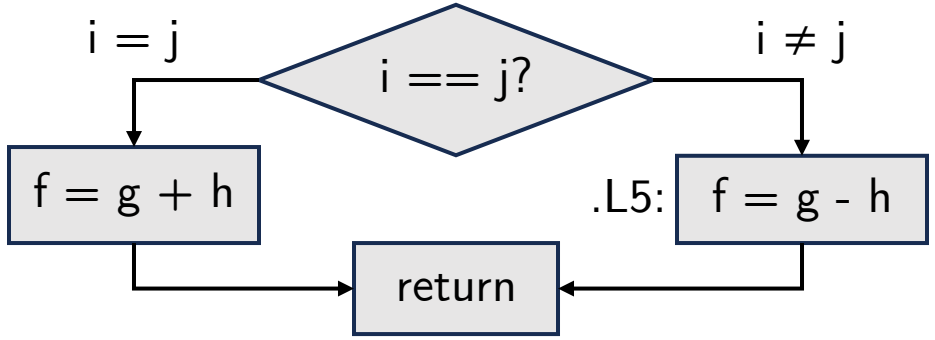
31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

- 6 base instruction formats: all **32-bit** wide:
  - opcode: 7-bit, specifies the operation.
  - rs1: 5-bit, register file address of the first source operand.
  - rs2: 5-bit, register file address of the second source operand.
  - rd: 5-bit, register file address of the destination.
  - imm: 12-bit or 20-bit, immediate number field.
  - funct: 3-bit or 10-bit, function code augmenting the opcode.
- B/S and U/J only differs in the immediate number encoding.

Control Instructions

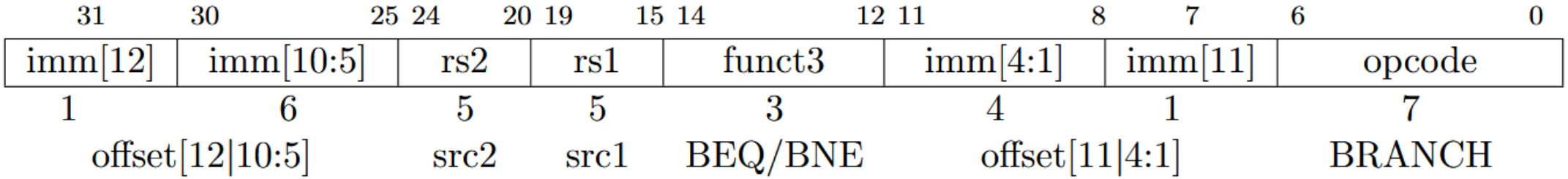
Branch Instructions

```
int foo(int i, int j, int g, int h) {
  int f;
  if (i == j) f = g + h; else f = g - h;
  return f;
}
```




```
# i: a0, j: a1, g: a2, h: a3, f: a0
foo(int, int, int, int):
        beq     a0,a1,.L5
        sub     a0,a2,a3
        ret
.L5:
        add     a0,a2,a3
        ret
```

- beq rs1, rs2, L1
  - Jumps to L1 if rs1 == rs2.
  - B-type instruction.
  - L1 encoded as 13-bit offset to the branch inst.
  - 13-bit means ± 4KiB range.
  - But immediate only has 12-bit?



Example

- All registers initialized to zero.
- What is the final value of a0?

- Online RISCV Interpreter:
  - https://www.cs.cornell.edu/courses/cs3410/2019sp/riscv/interpreter/

```
_start:
    xori a0, a0, 1
    xori a1, a1, 1
    xori t0, t0, 20
    xori t1, t1, 23
    bne t0, t1, inst1
    addi a0, a0, 1
    beq t0, t1, inst2
inst1:
    addi a0, a0, 2
    bne t0, zero, end
inst2:
    addi a0, a0, 3
end:
    sub a0, a0, a1
```

More Branch Conditions

- We have beq/bne, what about other conditions, e.g., less than?

- `slt rd, rs1, rs2`
  - If rs1 $<$ rs2, rd $=$ 1; Else rd $=$ 0.
- Other variants:
  - `slti  rd, rs1, 25`
  - `sltu  rd, rs1, rs2   # Compare as unsigned value.`
  - `sltui rd, rs1, 25`

- `blt rs1, rs2, Label` is a pseudo-instruction expanded by assembler:
  - `slt t0, rs1, rs2`
  - `bne t0, zero, Label  # branch if t0 != 0 -> t0 = 1 -> rs1 < rs2`

Bounds Check Shortcut

- Checking against array boundary $0 \leq x < y$ is very common!
- We can treat x, y as unsigned to check for $0 \leq x < y$ with one comparison.
  - `sltu t0, s1, t2  # t0 = 0 if`
  - `# s1 >= t2 (overflow) or`
  - `# s1 < 0 (underflow)`

- In two's complement representation, negative numbers look like large numbers in unsigned notation.
- Thus, an unsigned comparison of x < y also checks if x is negative as well as if x is less than y.

Loops

```
int foo(int *save, int k) {
    int i = -1;
    do i += 1; while (save[i] == k);
    return i;
}
```

```
# i: a0, k: a1, save[i]: a4, &save[i]: a5
foo(int*, int):
        mv      a5,a0
        li      a0,-1
.L2:
        addi    a0,a0,1
        addi    a5,a5,4
        lw      a4,-4(a5)
        beq     a4,a1,.L2
        ret
```

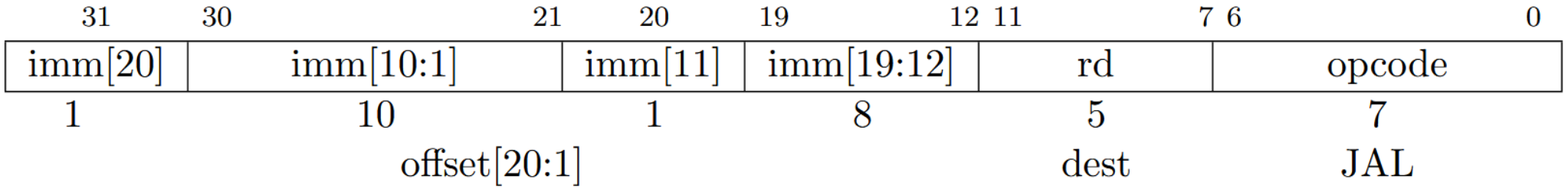
- a0 stores the induction variable i.
- a5 stores the address of save[i].
- a1 stores the compared variable k.
- Each iteration:
  - Increment i, &save[i].
  - Load save[i].
  - Compare save[i] with k.
  - Break out of the loop if save[i] != k.

Unconditional Jump

- RISC-V also has an unconditional branch instruction or jump instruction:

```
J-type:     jal rd, Label
```

- Jump and link (jal):
  - Jump to Label (21-bit offset with 20-bit intermediate → ± 1MiB range from PC).
  - Link: save PC + 4 in rd as the return address (see function calls later).

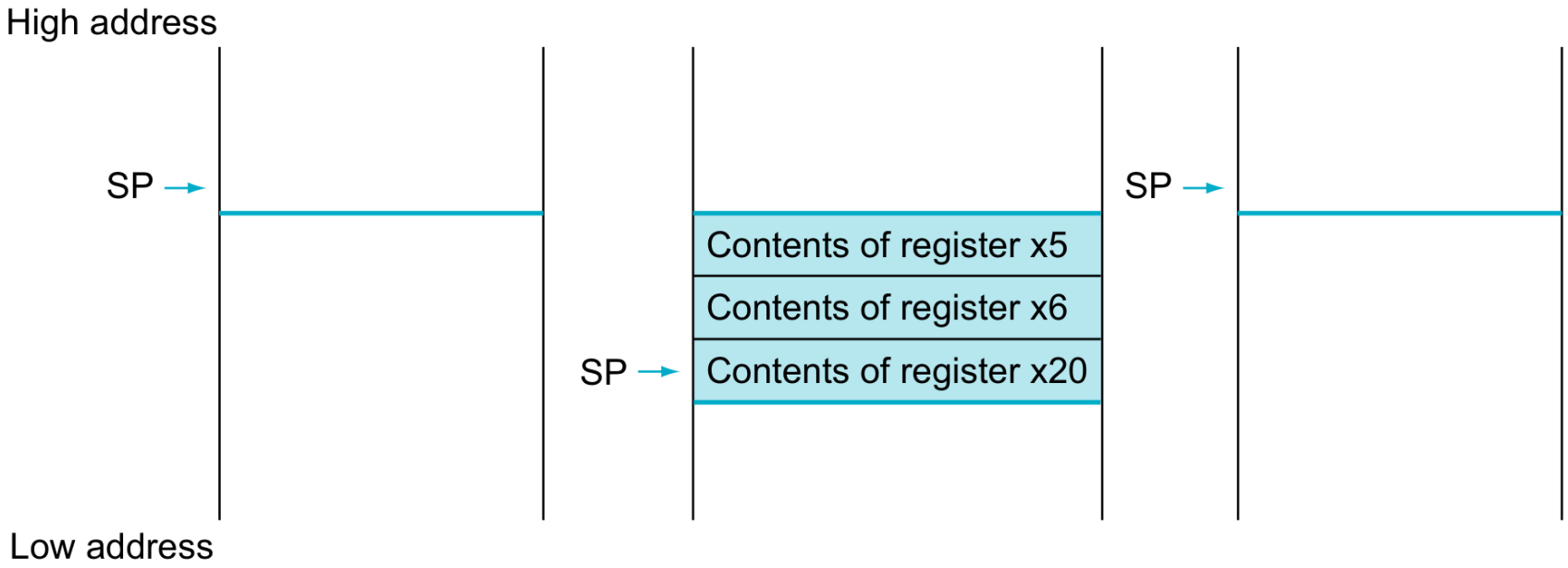


  - `j Label` is a pseudo-instruction that discard the return address (with rd = zero).
  - If conditional branch target cannot fit in 13-bit offset, you can rewrite with beq + jal.

Function Call

Stack

- Stack: A special region of memory used as a last-in-first-out queue.
  - Stack is accessed via address and load/store instructions, just like normal memory.
  - sp (stack pointer) register holds the address of the top of the stack.
  - Grows downwards – subtract sp to allocate, increment sp to deallocate.
- A handy temporary storage for values cannot fit in the register file.
  - We only have 32 registers :)



Function Calls

- One function (caller) calls another function (callee).
  - Resources are shared between two functions, e.g., registers, memory.
- We need some convention between caller and callee function to:
  - Pass arguments, allocate registers/stack, get return value, etc.

```
__attribute__((noinline))
int callee(int a, int b) {
    return a + b;
}
int caller() {
    int a = 2, b = 3;
    int ret = callee(a, b);
    ret += 8;
    return ret;
}
```

```
callee:
0x00    add     a0,a0,a1
0x04    jalr    zero,0(ra)
caller:
0x08    addi    sp,sp,-16
0x0C    li      a1,3
0x10    li      a0,2
0x14    sw      ra,12(sp)
0x18    jal     ra, callee
0x1C    lw      ra,12(sp)
0x20    addi    a0,a0,8
0x24    addi    sp,sp,16
0x28    jalr    zero,0(ra)
```

```
Register File:
ra:     0x1234_5678
sp:     0x0000_00A0
a0:     0x0000_0000
a1:     0x0000_0000

Stack:
sp -> 0x0A0:    0000_0000
      0x09C:    0000_0000
      0x098:    0000_0000
      0x094:    0000_0000
      0x090:    0000_0000
```

Steps of Function Call

- Step 1: Caller sets up the arguments.
  - a0 – a7 holds the first 8 arguments.
  - Here we have two arguments: a0 holds a = 2; a1 holds b = 3.
  - li is a pseudo-instruction to load all 32-bit intermediate.

```
__attribute__((noinline))
int callee(int a, int b) {
    return a + b;
}
int caller() {
    int a = 2, b = 3;
    int ret = callee(a, b);
    ret += 8;
    return ret;
}
```

```
callee:
0x00    add     a0,a0,a1
0x04    jalr    zero,0(ra)
caller:
0x08    addi    sp,sp,-16
0x0C    li      a1,3
0x10    li      a0,2
0x14    sw      ra,12(sp)
0x18    jal     ra, callee
0x1C    lw      ra,12(sp)
0x20    addi    a0,a0,8
0x24    addi    sp,sp,16
0x28    jalr    zero,0(ra)
```

```
Register File:
ra:     0x1234_5678
sp:     0x0000_0090
a0:     0x0000_0002
a1:     0x0000_0003

Stack:
        0x0A0:   0000_0000
        0x09C:   0000_0000
        0x098:   0000_0000
        0x094:   0000_0000
sp -> 0x090:   0000_0000
```

Steps of Function Call

- Step 2: Caller transfers the control to callee.
  - Save the return address in ra, jump to callee.
  - Return address: address of next instruction to execute once callee returns (PC + 4).
  - Here, ra is overwritten to 0x1C.

```
__attribute__((noinline))
int callee(int a, int b) {
    return a + b;
}
int caller() {
    int a = 2, b = 3;
    int ret = callee(a, b);
    ret += 8;
    return ret;
}
```

```
callee:
0x00    add     a0,a0,a1
0x04    jalr    zero,0(ra)
caller:
0x08    addi    sp,sp,-16
0x0C    li      a1,3
0x10    li      a0,2
0x14    sw      ra,12(sp)
0x18    jal     ra, callee
0x1C    lw      ra,12(sp)
0x20    addi    a0,a0,8
0x24    addi    sp,sp,16
0x28    jalr    zero,0(ra)
```

```
Register File:
ra:     0x0000_001C
sp:     0x0000_0090
a0:     0x0000_0002
a1:     0x0000_0003

Stack:
        0x0A0:   0000_0000
        0x09C:   0000_0000
        0x098:   0000_0000
        0x094:   0000_0000
sp -> 0x090:   0000_0000
```

Steps of Function Call

- Step 3: Callee perform the computation.
  - Get arguments from a0 – a7 → Perform the computation → Store result in a0.
  - Here we set a0 = a0 + a1 = a + b.

```
__attribute__((noinline))
int callee(int a, int b) {
    return a + b;
}
int caller() {
    int a = 2, b = 3;
    int ret = callee(a, b);
    ret += 8;
    return ret;
}
```

```
callee:
0x00    add     a0,a0,a1
0x04    jalr    zero,0(ra)
caller:
0x08    addi    sp,sp,-16
0x0C    li      a1,3
0x10    li      a0,2
0x14    sw      ra,12(sp)
0x18    jal     ra, callee
0x1C    lw      ra,12(sp)
0x20    addi    a0,a0,8
0x24    addi    sp,sp,16
0x28    jalr    zero,0(ra)
```

```
Register File:
ra:     0x0000_001C
sp:     0x0000_0090
a0:     0x0000_0005
a1:     0x0000_0003

Stack:
        0x0A0:   0000_0000
        0x09C:   0000_0000
        0x098:   0000_0000
        0x094:   0000_0000
sp ->   0x090:   0000_0000
```

Steps of Function Call

- Step 4: Callee transfers back to caller.
  - Jump to the return address stored in ra.
  - Here the return address is 0x1C, which is the instruction after the calling jal.

```
__attribute__((noinline))
int callee(int a, int b) {
    return a + b;
}
int caller() {
    int a = 2, b = 3;
    int ret = callee(a, b);
    ret += 8;
    return ret;
}
```

```
callee:
0x00    add     a0,a0,a1
0x04    jalr    zero,0(ra)
caller:
0x08    addi    sp,sp,-16
0x0C    li      a1,3
0x10    li      a0,2
0x14    sw      ra,12(sp)
0x18    jal     ra, callee
0x1C    lw      ra,12(sp)
0x20    addi    a0,a0,8
0x24    addi    sp,sp,16
0x28    jalr    zero,0(ra)
```

```
Register File:
ra:     0x0000_001C
sp:     0x0000_0090
a0:     0x0000_0005
a1:     0x0000_0003

Stack:
        0x0A0:   0000_0000
        0x09C:   0000_0000
        0x098:   0000_0000
        0x094:   0000_0000
sp -> 0x090:   0000_0000
```

Steps of Function Call

- Step 5: Callee continues execution.
  - Note the result is stored in a0.

```
__attribute__((noinline))
int callee(int a, int b) {
    return a + b;
}
int caller() {
    int a = 2, b = 3;
    int ret = callee(a, b);
    ret += 8;
    return ret;
}
```

```
callee:
0x00    add     a0,a0,a1
0x04    jalr    zero,0(ra)
caller:
0x08    addi    sp,sp,-16
0x0C    li      a1,3
0x10    li      a0,2
0x14    sw      ra,12(sp)
0x18    jal     ra, callee
0x1C    lw      ra,12(sp)
0x20    addi    a0,a0,8
0x24    addi    sp,sp,16
0x28    jalr    zero,0(ra)
```

```
Register File:
ra:     0x0000_001C
sp:     0x0000_0090
a0:     0x0000_0005
a1:     0x0000_0003

Stack:
        0x0A0:   0000_0000
        0x09C:   0000_0000
        0x098:   0000_0000
        0x094:   0000_0000
sp -> 0x090:   0000_0000
```

Handle Register Conflict

- What if caller and callee used the same register?
  - Caller is also a function, has the return address in ra.
  - But jal will overwrite the ra.
  - Now when caller returns, it will jump back to itself (0x1C)!

```
__attribute__((noinline))
int callee(int a, int b) {
    return a + b;
}
int caller() {
    int a = 2, b = 3;
    int ret = callee(a, b);
    ret += 8;
    return ret;
}
```

```
callee:
0x00    add     a0,a0,a1
0x04    jalr    zero,0(ra)
caller:
0x08    addi    sp,sp,-16
0x0C    li      a1,3
0x10    li      a0,2
0x14    sw      ra,12(sp)
0x18    jal     ra, callee
0x1C    lw      ra,12(sp)
0x20    addi    a0,a0,8
0x24    addi    sp,sp,16
0x28    jalr    zero,0(ra)
```

```
Register File:
ra:     0x0000_001C
sp:     0x0000_0090
a0:     0x0000_0005
a1:     0x0000_0003

Stack:
        0x0A0:   0000_0000
        0x09C:   0000_0000
        0x098:   0000_0000
        0x094:   0000_0000
sp -> 0x090:   0000_0000
```

Save Conflict Register on Stack

- Caller/callee views each other as a black-box.
  - Conservative, assuming the other would use all registers.
  - Save potential conflict registers on stack.
  - Caller-save: If used, the caller needs to: save it → call callee → restore it.
  - Callee-save: If used, the callee needs to: save it → use it → restore it → return.

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	—
x3	gp	Global pointer	—
x4	tp	Thread pointer	Callee
x5	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP args/return values	Caller
f12-17	fa2-7	FP args	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

Example: Caller saves RA

- Step 1: Allocate some space on stack.
  - Recall that stack grows downwards → subtraction from sp allocates stack.

```
__attribute__((noinline))
int callee(int a, int b) {
    return a + b;
}
int caller() {
    int a = 2, b = 3;
    int ret = callee(a, b);
    ret += 8;
    return ret;
}
```

```
callee:
0x00    add     a0,a0,a1
0x04    jalr    zero,0(ra)
caller:
0x08    addi    sp,sp,-16
0x0C    li      a1,3
0x10    li      a0,2
0x14    sw      ra,12(sp)
0x18    jal     ra, callee
0x1C    lw      ra,12(sp)
0x20    addi    a0,a0,8
0x24    addi    sp,sp,16
0x28    jalr    zero,0(ra)
```

```
Register File:
ra:     0x1234_5678
sp:     0x0000_0090
a0:     0x0000_0000
a1:     0x0000_0000

Stack:
        0x0A0:   0000_0000
        0x09C:   0000_0000
        0x098:   0000_0000
        0x094:   0000_0000
sp ->   0x090:   0000_0000
```

Step 2: Caller saves RA

- Step 2: Save ra on stack before function call.
  - Recall that stack is just special region of memory, accessed by load/store instructions.
  - Using a sw (store_word) instruction and sp as the address.
  - Here, sp + 12 = 0x90 + 12 = 0x9C.

```
__attribute__((noinline))
int callee(int a, int b) {
    return a + b;
}
int caller() {
    int a = 2, b = 3;
    int ret = callee(a, b);
    ret += 8;
    return ret;
}
```

```
callee:
0x00    add     a0,a0,a1
0x04    jalr    zero,0(ra)
caller:
0x08    addi    sp,sp,-16
0x0C    li      a1,3
0x10    li      a0,2
0x14    sw      ra,12(sp)
0x18    jal     ra, callee
0x1C    lw      ra,12(sp)
0x20    addi    a0,a0,8
0x24    addi    sp,sp,16
0x28    jalr    zero,0(ra)
```

```
Register File:
ra:     0x1234_5678
sp:     0x0000_0090
a0:     0x0000_0000
a1:     0x0000_0000

Stack:
        0x0A0:   0000_0000
        0x09C:   1234_5678
        0x098:   0000_0000
        0x094:   0000_0000
sp -> 0x090:   0000_0000
```

Example: Caller saves RA

- Step 3: Restore ra after function call.
  - Recall that stack is just special region of memory, accessed by load/store instructions.
  - Using a lw (load_word) instruction and sp to read back the saved return address from ra.
  - Now ra is restored to the original value 0x1234_5678.
  - And the caller can return normally.

```
__attribute__((noinline))
int callee(int a, int b) {
    return a + b;
}
int caller() {
    int a = 2, b = 3;
    int ret = callee(a, b);
    ret += 8;
    return ret;
}
```

```
callee:
0x00    add     a0,a0,a1
0x04    jalr    zero,0(ra)
caller:
0x08    addi    sp,sp,-16
0x0C    li      a1,3
0x10    li      a0,2
0x14    sw      ra,12(sp)
0x18    jal     ra, callee
0x1C    lw      ra,12(sp)
0x20    addi    a0,a0,8
0x24    addi    sp,sp,16
0x28    jalr    zero,0(ra)
```

```
Register File:
ra:     0x1234_5678
sp:     0x0000_0090
a0:     0x0000_0000
a1:     0x0000_0000

Stack:
        0x0A0:   0000_0000
        0x09C:   1234_5678
        0x098:   0000_0000
        0x094:   0000_0000
sp ->   0x090:   0000_0000
```

Support More than 8 Arguments

- a0 – a7 can hold 8 arguments → Extra arguments are passed through stack!

```
__attribute__((noinline))
int callee(int a, int b, int c, int d,
           int e, int f, int g, int h,
           int extra) {
    return a + b + c + d + e
         + f + g + h + extra;
}
int caller() {
    int a = 2, b = 3, c = 4, d = 5;
    int e = 6, f = 7, g = 8, h = 9;
    int extra = 10;
    int ret = callee(
        a, b, c, d, e, f, g, h,
        extra);
    ret += 8;
    return ret;
}
```

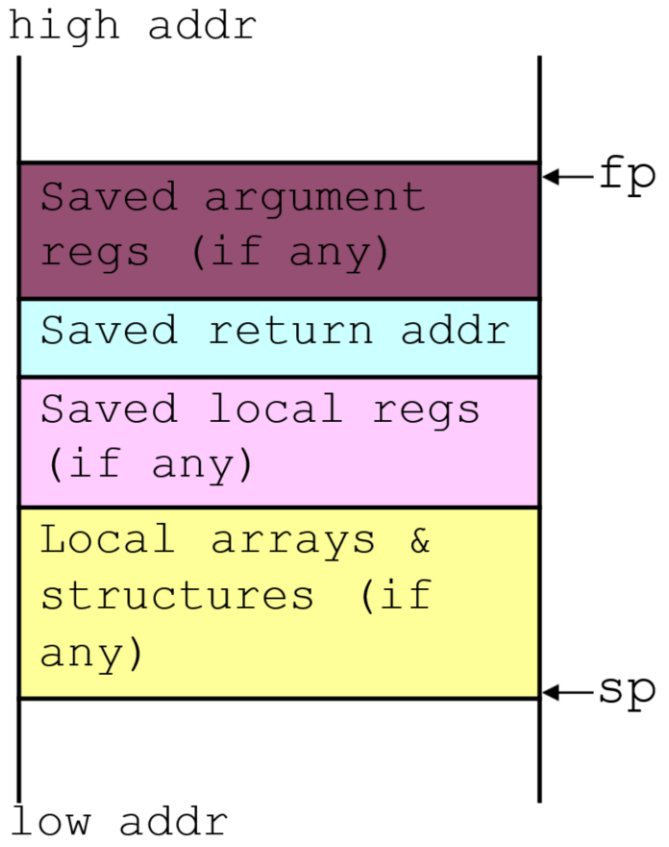
```
_Z6calleeiiiiiiiii:
        ...
        lw      a5,0(sp)
        add     a0,a0,a5
        ret
_Z6callerv:
        addi    sp,sp,-32
        li      a5,10
        sw      a5,0(sp)
        ...
        li      a0,2
        sw      ra,28(sp)
        call    _Z6calleeiiiiiiiii
        lw      ra,28(sp)
        addi    a0,a0,8
        addi    sp,sp,32
        jr      ra
```

*Read **extra** from stack.*

*Save **extra** to stack.*

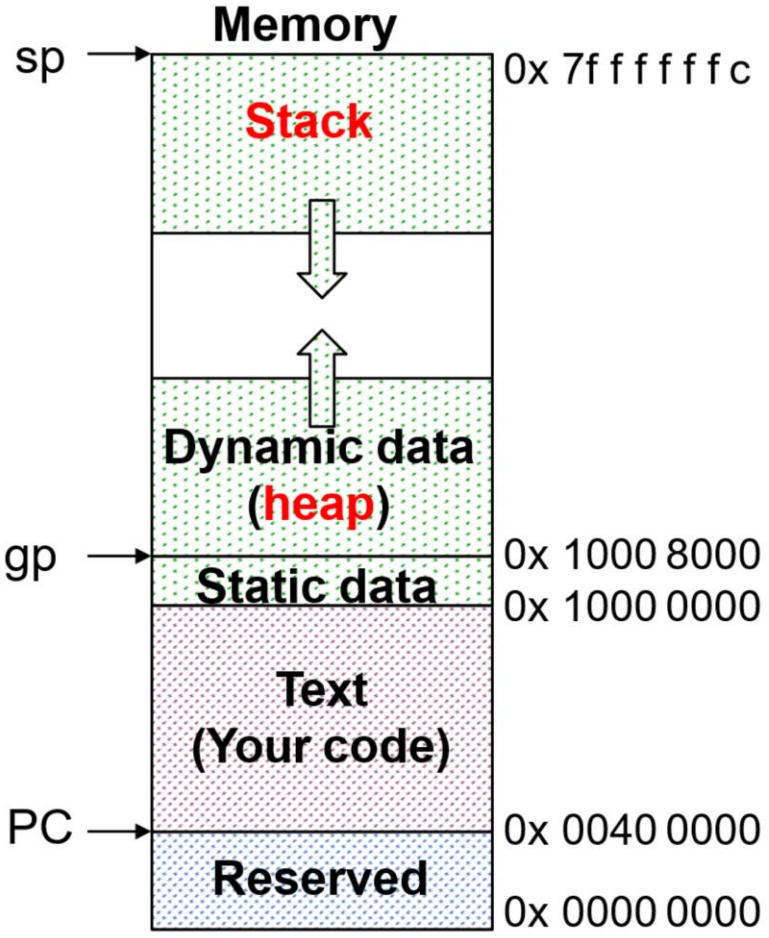
Stack Frame

- Stack holds local values cannot fit in the register file.
- Imagine each function has a frame holds local data:
- Stack is last-in-first-out – perfect match to hold frames:
  - Call a function → push a frame on the stack.
  - Returning from a function → pop a frame from the stack.
- The frame pointer (fp) points to the first word of the frame of a function – providing a stable "base" register for the function.
- fp is initialized using sp on a call and sp is restored using fp on a return.



Allocating Space on the Heap

- Static data segment for constants and other static variables (e.g., global arrays).
- Dynamic data segment (aka heap) for structures that grow and shrink (e.g., linked lists).
- Allocate space on the heap with malloc() and free it with free() in C.



Example: Compiling a Recursive Function

- Some classification on function type:
  - Leaf function does not call other function.
  - Nested function calls other function.
  - Recursive function calls itself.

- Computer the factorial of n = n x (n-1) x ... 1.
  - factor(0) = 1
  - factor(1) = 1 * factor(0) = 1
  - factor(2) = 2 * factor(1) = 2
  - factor(3) = 3 * factor(2) = 6
  - ...

```
int factor(int n) {
    if (n == 0)
        return 1;
    else
        return n * factor(n - 1);
}
```

Compile factor()

```c
int factor(int n) {
    if (n == 0)
        return 1;
    else
        return n * factor(n - 1);
}
```

- Notes:
  - jr ra is pseudo-instruction for jalr zero, 0(ra)
  - call label is pseudo-instruction for jal ra, label
  - j label is pseudo-instruction for jal zero, label
  - ret is pseudo-instruction for jalr zero, 0(ra)

```
factor:
        addi    sp,sp,-32
        sw      ra,28(sp)
        sw      s0,24(sp)
        addi    s0,sp,32
        sw      a0,-20(s0)
        lw      a5,-20(s0)
        bne     a5,zero,.L2
        li      a5,1
        j       .L3
.L2:
        lw      a5,-20(s0)
        addi    a5,a5,-1
        mv      a0,a5
        call    factor
        mv      a4,a0
        lw      a5,-20(s0)
        mul     a5,a4,a5
.L3:
        mv      a0,a5
        lw      ra,28(sp)
        lw      s0,24(sp)
        addi    sp,sp,32
        jr      ra
```

Practice

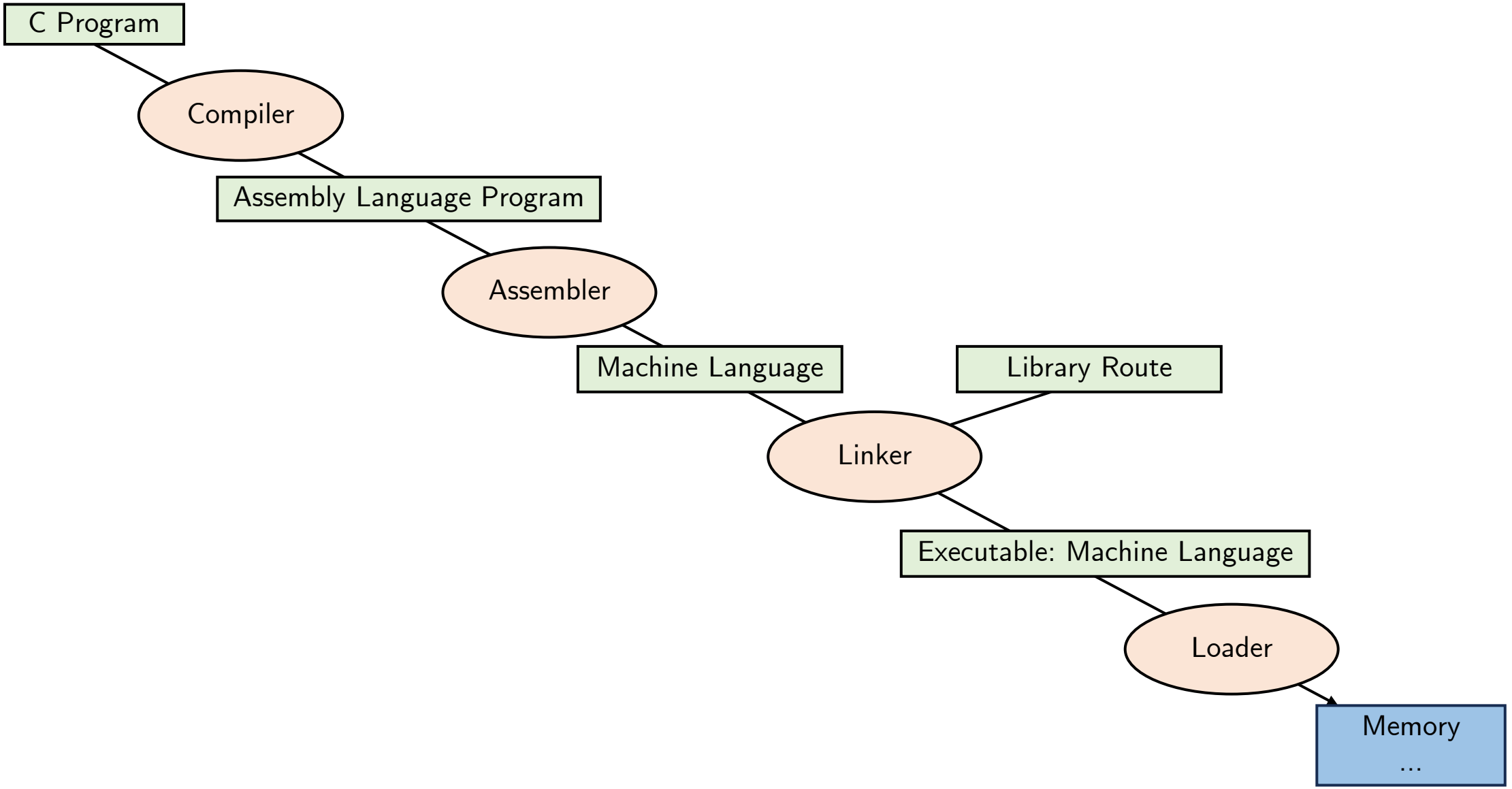
- Give the assembly code of the following function:
  - a0 has g, a1 has h, a2 has i, a3 has j.

```
int foo(int g, int h, int i, int j) {
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

```
foo:
        add     a0,a0,a1
        add     a2,a2,a3
        sub     a0,a0,a2
        ret
```

Summary

Recap C Compilation Flow



Compile Benefit

- Compiler can aggressively optimize the generated machine code.
  - Controlled by –Ox to specify optimization level. –O0 no optimization; -O3 aggressive.
- Example optimizations:
  - Constant propagation, function inline, loop unroll, vectorization, etc.

```
// Source code
int foo(int *array, int N) {
    int ret = 0;
    for (int i = 0; i < N; ++i)
        ret += array[i];
    return ret;
}
int main() {
    int a[2] = {0, 2};
    return foo(a, 2);
}
```

```
# -O1
main:
        addi    sp,sp,-32
        sw      ra,28(sp)
        sw      zero,8(sp)
        li      a5,2
        sw      a5,12(sp)
        mv      a1,a5
        addi    a0,sp,8
        call    foo
        lw      ra,28(sp)
        addi    sp,sp,32
        jr      ra
```

```
# -O2
main:
        li      a0,2
        ret
```

Compile Benefit

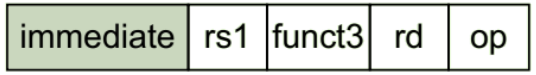
- Delivers higher performance.

gcc optimization	Relative performance	Clock cycles (millions)	Instruction count (millions)	CPI
None	1.00	158,615	114,938	1.38
01 (medium)	2.37	66,990	37,470	1.79
02 (full)	2.38	66,521	39,993	1.66
03 (procedure integration)	2.41	65,747	44,993	1.46

FIGURE 2.26 Comparing performance, instruction count, and CPI using compiler optimization for Bubble Sort. The programs sorted 100,000 32-bit words with the array initialized to random values. These programs were run on a Pentium 4 with a clock rate of 3.06 GHz and a 533 MHz system bus with 2 GB of PC2100 DDR SDRAM. It used Linux version 2.4.20.

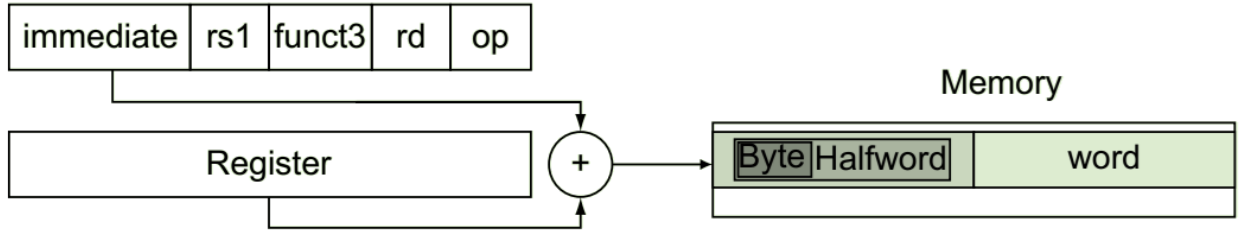
Addressing Mode

1. Immediate addressing



2. Register addressing

3. Base addressing



4. PC-relative addressing