Module 06: Instruction Set Architecture, RISC-V Assembly Programming, and Assembly Program of a C Program

Units 4 and 5: Supporting functions and procedures, sorting example and comparison with other ISAs

ITSC 2181 - Introduction to Computer Systems

College of Computing and Informatics

COLLEGE OF COMPUTING AND INFORMATICS

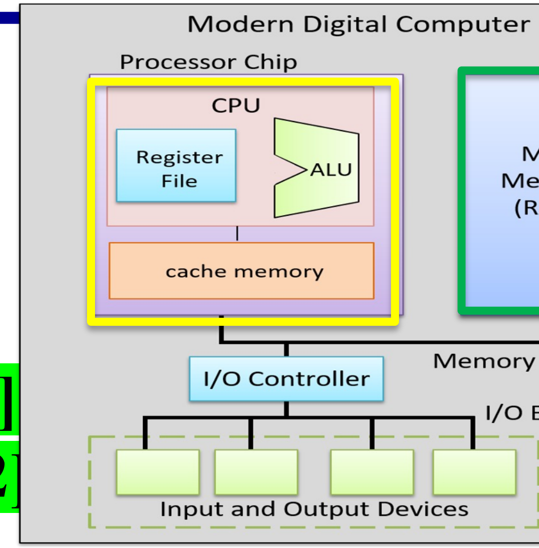
Module 06: Instruction Set Architecture, RISC-V Assembly Programming, and Assembly Program of a C Program

- **Unit 1: Module overview, Instruction Set Architecture (ISA) and assembly programs, registers, instruction operations and operands, register and immediate operands, arithmetic and logic instructions**
- **Unit 2: Memory Operands and Memory Access Instructions**
- **Unit 3: Conditional control instructions for making decisions (if-else) and loops**
- ☛ **Unit 4: Supporting Functions and procedures**
- ☛ **Unit 5: Sort examples and comparison with other ISAs**

- **Materials are developed based on textbook:**
  - Computer Organization and Design RISC-V Edition: The Hardware/Software Interface, Amazon
  - RISC-V Specification: https://riscv.org/technical/specifications/
  - ITSC 3181: https://passlab.github.io/ITSC3181/

Instructions Used So Far: add, addi, sub, slli, load, store, and beq/bne/bge/blt

```
add  x10, x5, x6          // [x10] = [x5] + [x6]
addi x10, x5, 100         // [x10] = [x5] + 100
sub  x11, x5, x6          // [x11] = [x5] – [x6]
slli  x12, x5, 5          // [x12] = x5 * 2^^5
ld x12, 32(x5)            // [x12] = Mem[32 + [x5]]
sd x12, 32(x5)            // Mem[32 + [x5]] = [x12]
beq x5, x6, <label1> // if ([x5] == [x6]) …
```



- They can do computation and access memory, and implementing complicated computation and algorithms involving decision making and repetive
- Organizing software to make them modular and easily reusable
  – Function and function call (procedure, method, etc)

Three Kinds of Operands and Three Classes of Instructions

- **General form:**
  - <op word> <dest operand> <src operand 1> <src operand 2>
  - E.g.: add x5, x3, x4, which performs [x5] = [x3] + [x4]

Three Kinds of Operands

1. Register operands, e.g., x0 – x31
2. Immediate operands, e.g., 0, -10, etc
3. Memory operands, e.g. 16(x4)

Module 06: Unit 1

Module 06: Unit 2

Module 06: Unit 3

Three Classes of Instructions

1. Arithmetic-logic instructions
   - add, sub, addi, and, or, shift left|right, etc
2. Memory load and store instructions
   - lw and sw: Load/store word
   - ld and sd: Load/store doubleword
3. Control transfer instructions (changing sequence of instruction execution)
   - Conditional branch: bne, beq
   - Unconditional jump: j (
   - **Procedure call and return: jal and jr**

Function Call: sum_full.c

sum_full.c

```c
REAL sum(int N, REAL X[], REAL a) {
    int i;
    REAL result = 0.0;
    for (i = 0; i < N; ++i)
        result += a * X[i];
    return result;
}
```

sum_full.c

```c
    srand48((1 << 12));
    init(X, N);
    init(Y, N);
    REAL a = 0.1234;
    /* example run */
    elapsed = read_timer();
    REAL result = sum(N, X, a);
    elapsed = (read_timer() - elapsed);
```

https://passlab.github.io/ITSC3181/exercises/sum

Function Call Steps

1. Place arguments for callee in registers
2. Transfer control to callee function
3. Acquire storage for callee function
4. Perform callee's operations
5. Place result in register for caller
6. Return to place of call

```c
REAL sum(int N, REAL X[], REAL a) {
    int i;
    REAL result = 0.0;
    for (i = 0; i < N; ++i)
        result += a * X[i];
    return result;
}
```

```c
    srand48((1 << 12));
    init(X, N);
    init(Y, N);
    REAL a = 0.1234;
    /* example run */
    elapsed = read_timer();
    REAL result = sum(N, X, a);
    elapsed = (read_timer() - elapsec
```

Three Important Things of the Computer System to Support Function Calls

1. **Hardware instructions for control transfer for procedure call and call return**
   - Caller→callee transfer
   - Callee→caller transfer
2. **Specifying register/memory for passing data between caller and callee**
   - Passing argument from caller→callee
   - Passing return value from callee →caller
3. **Mechanism of stack memory for managing data of functions**
   - Storage for function variables, etc
   - Preserve register data of the caller when control is in callee
   - Restore the data when control is returned to caller

```c
REAL sum(int N, REAL X[], REAL a) {
    int i;
    REAL result = 0.0;
    for (i = 0; i < N; ++i)
        result += a * X[i];
    return result;
}
```

```c
    srand48((1 << 12));
    init(X, N);
    init(Y, N);
    REAL a = 0.1234;
    /* example run */
    elapsed = read_timer();
    REAL result = sum(N, X, a);
    elapsed = (read_timer() - elapse
```

Sum Example, sum_full_riscv.s

```
    return result;
```

```
        .globl  sum
        .type   sum, @functio
sum:
        addi    sp,sp,-48
        sd      s0,40(sp)
        addi    s0,sp,48
        mv      a5,a0
        sd      a1,-48(s0)
        fsw     fa0,-40(s0)
        sw      a5,-36(s0)
```

```
        sext.w  a5,a5
        blt     a4,a5,.L10
        flw     fa5,-24(s0)
        fmv.s   fa0,fa5
        ld      s0,40(sp)
        addi    sp,sp,48
        jr      ra
```

Return to caller with return value stored in register fa0

```
REAL result = sum(N, X, a);
```

```
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-80
        sd      ra,72(sp)
        sd      s0,64(sp)
        addi    s0,sp,80
        mv      a5,a0
```

```
        fsw     fa5,-44(s0)
        call    read_timer
        fsd     fa0,-56(s0)
        lw      a5,-20(s0)
        flw     fa0,-44(s0)
        ld      a1,-32(s0)
        mv      a0,a5
        call    sum
        fsw     fa0,-60(s0)
        call    read_timer
```

Args for sum call in reg a0, fa0, a5

Store return address in reg x1 and call transfer to sum

https://passlab.github.io/ITSC3181/exercises/sum

1. Hardware Instruction for Function Call

- Function call: jump and link

  `jal x1, ProcedureLabel`

  - Address of following instruction put in x1
  - Jumps to target address

- Function return: jump and link register

  `jalr x0, 0(x1)`

  - Like jal, but jumps to 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

In Summary for jal and jalr Instructions

- The jal (jump and link) instruction in RISC-V is used for making function calls. It jumps to the target function's address while saving the return address in the link register (ra). Function arguments can be passed in registers before the jal instruction. The jalr (jump and link register) instruction is used for function call returns, where it jumps to the address stored in the link register, returning control to the calling function at the point just after the original jal instruction. Together, these instructions enable function calls and returns in RISC-V assembly language.

2. Register Usage *Convention* for Function Call

- x10 – x17: arguments and return values for function calls (a0 – a17)
  - https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf
  - https://inst.eecs.berkeley.edu/~cs61c/resources/RISCV_Calling_Convention.pdf
- x5 – x7, x28 – x31: temporary registers (t0-t6)
  - Not automatically preserved by the callee
- x8 – x9, x18 – x27: saved registers (s0-s11)
  - If used, the callee saves and restores them

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Table 18.2: RISC-V calling convention register usage.

```
sult = sum(N, X, a);

    .globl   main
    .type    main, @function

    addi     sp,sp,-80
    sd       ra,72(sp)
    sd       s0,64(sp)
    addi     s0,sp,80
    mv       a5,a0

    fsw      fa5,-44(s0)
    call     read_timer
    fsd      fa0,-56(s0)
    lw       a5,-20(s0)
    flw      fa0,-44(s0)
    ld       a1,-32(s0)
    mv       a0,a5
    call     sum
    fsw      fa0,-60(s0)
    call     read_timer
```

Args for sum call in reg a0, fa0, a5

Store return address in reg x1 and call transfer to sum

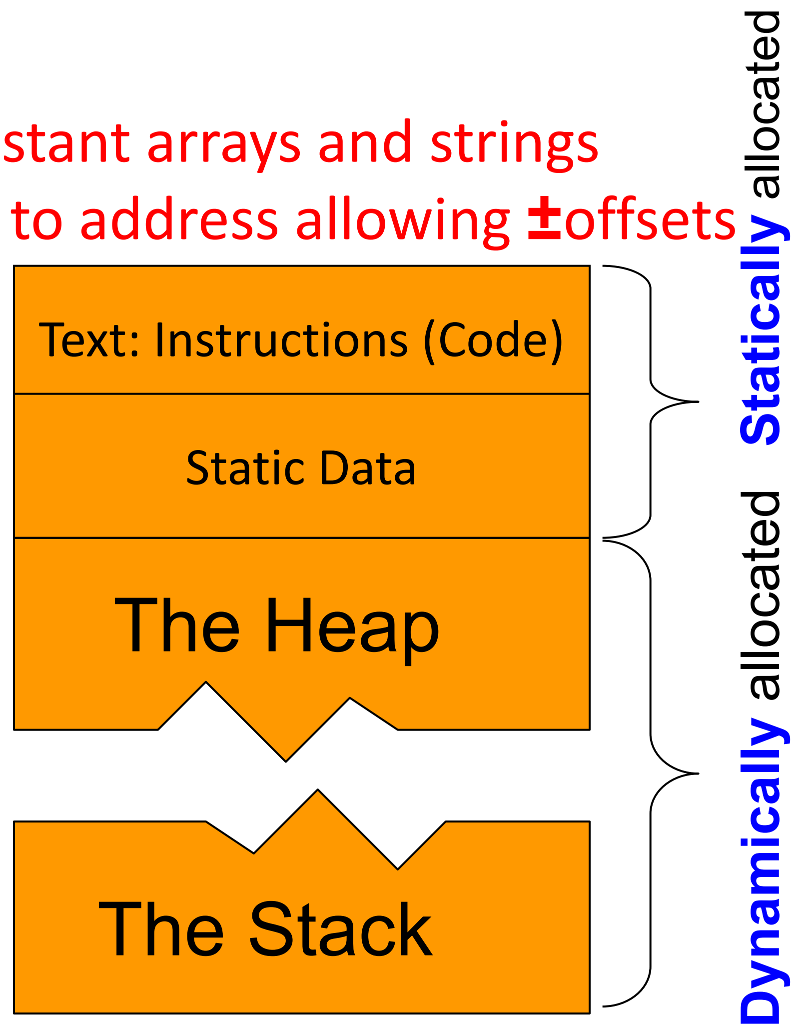
Register a0-a7, and s0-s11

1. a0-a7 Registers (Argument Registers):
   1. Purpose: The a0-a7 registers, also known as the argument registers, are primarily used to pass function arguments to a called function.
   2. Usage: When a function is called, arguments are typically placed in the a0-a7 registers before the jal (jump and link) instruction is executed. The called function can access these values directly from these registers.
   3. Saving Values: The called function should not assume that the argument values in a0-a7 will remain unchanged after the function call. If it needs to preserve or modify these values, it should save them to other registers or memory before overwriting them.
2. s0-s11 Registers (Saved Registers):
   1. Purpose: The s0-s11 registers, also known as the saved registers, are used for saving and preserving values across function calls. They are callee-saved registers, meaning that the called function must ensure their values are preserved across the function call and restore them before returning to the calling function.
   2. Usage: When a function is called, it must save the contents of the s0-s11 registers if it intends to modify these registers. This ensures that any values saved in these registers by the calling function are not inadvertently changed.
   3. Saving Values: To save the values of s0-s11, the callee (the called function) typically pushes these registers onto the stack in the function prologue (the beginning of the function). After the function has finished executing, it restores the saved values from the stack in the function epilogue (the end of the function) to ensure that the calling function's expectations are met regarding the values in these registers.

- In summary, the a0-a7 registers are used to pass function arguments, and the s0-s11 registers are used to save and preserve registers across function calls. Proper management of these registers is essential to ensure the correct and efficient execution of functions in a RISC-V assembly program

3. Stack Memory for Managing Data of Function Call

- Memory Layout of a Process
  - Text: program code
  - Static data: global variables
    - e.g., static variables in C, constant arrays and strings
    - x3 (global pointer) initialized to address allowing **±**offsets into this segment
  - Dynamic data: heap
    - E.g., malloc in C, new in Java
  - **Stack: automatic storage for function**
    - **Variables**
    - **For preserving data in registers**



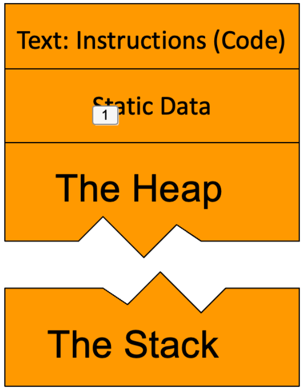
How Stack Works For Function Calls

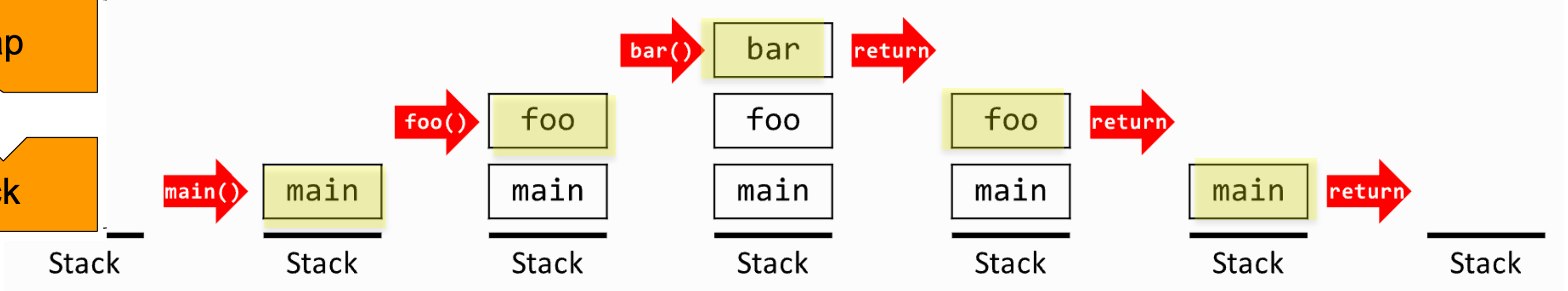
- Stack Memory for Each Function Call
  - **Named as Stack Frame, Function frame (activation record)**
  - Memory space for function's parameters and local variables, temporary objects, the return address, and other items that are needed by the function.

```
void bar() {
}

void foo() {
  bar();
}

int main() {
  foo();
}
```





https://eecs280staff.github.io/notes/02_ProceduralAbstraction_Testing.html

How Stack Works For Function Calls

```cpp
int plus_one(int x) {
  return x + 1;
}

int plus_two(int x) {
  return plus_one(x + 1);
}

int main() {
  int result = 0;
  result = plus_one(0);
  result = plus_two(result);
  cout << result;            // prints 3
}
```

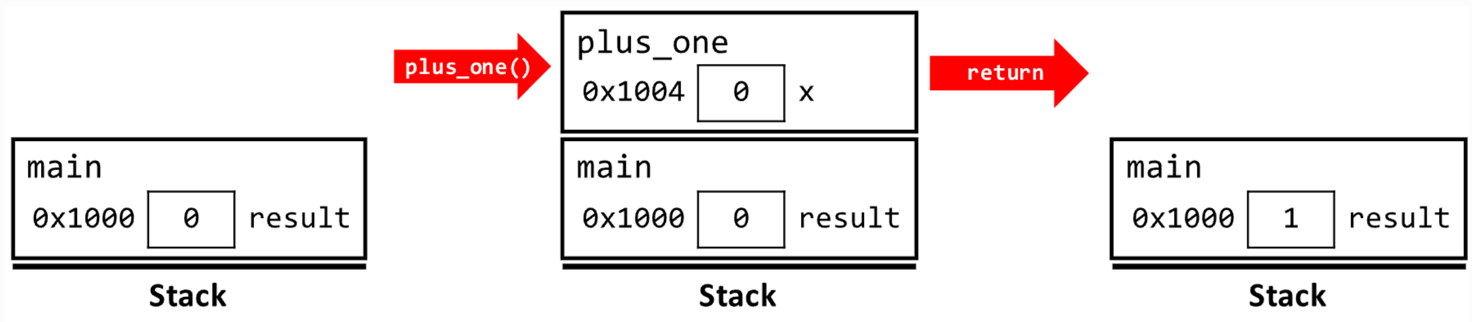


Figure 8 *Activation record for* `plus_one()`.

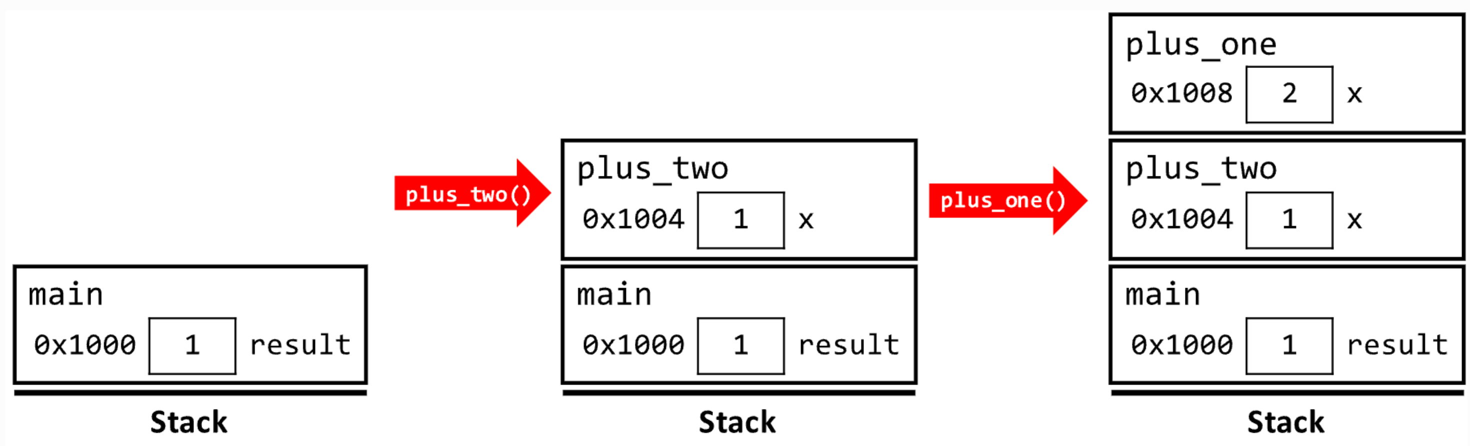
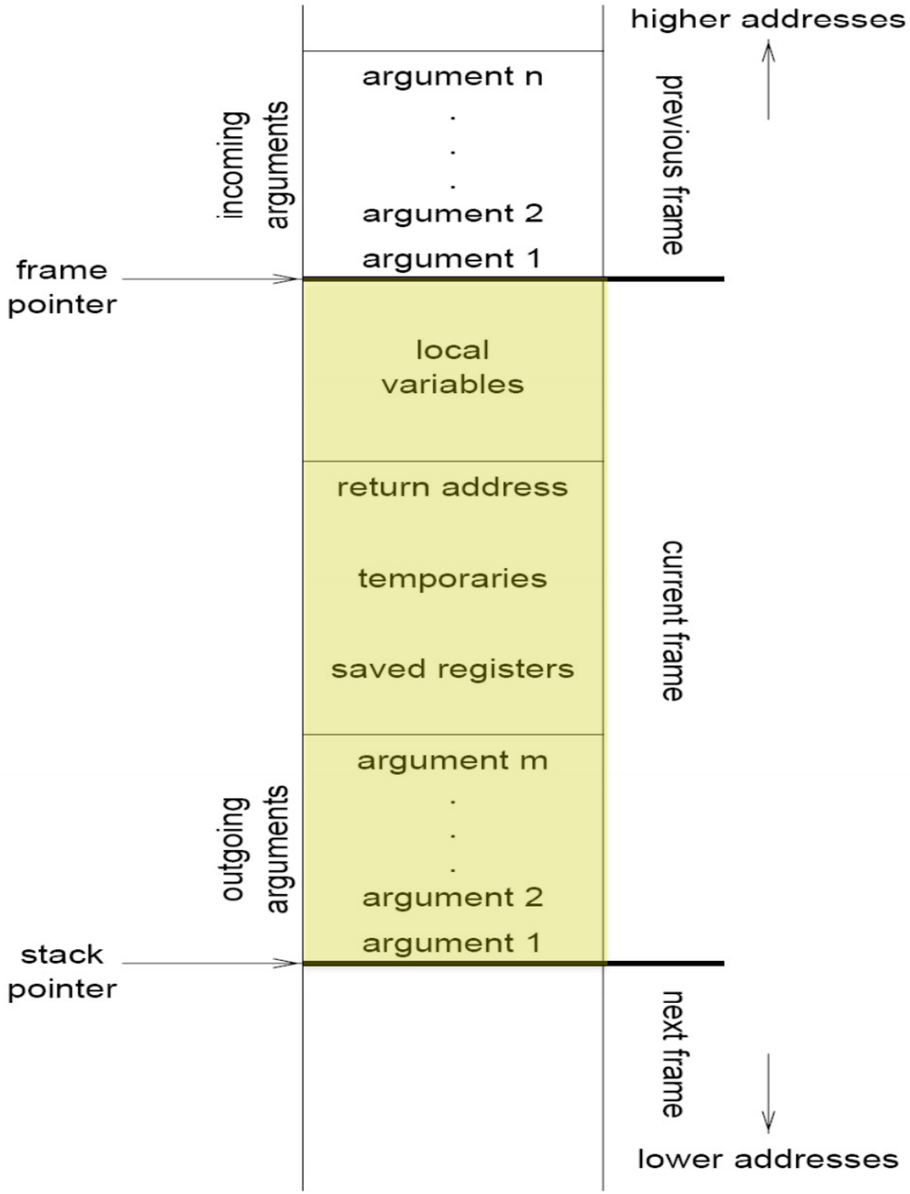


Figure 9 *State of stack in second call to* `plus_one()`.

Stack Frame (Activation Record) of a Function Call

- Information:
  - Parameters
  - Local variables
  - Return address
  - Location to put return value when function exits
  - Control link to the caller’s activation record
  - Saved registers
  - Temporary variables and intermediate results
  - (not always) Access link to the function’s static parent
- Frame pointer (fp register): the starting address of AR
- Stack pointer (sp register): the ending address of AR



Leaf Procedure Example

- Leaf procedure: a procedure does not call other procedures
  - Thinking of procedure calls as a tree

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

  - Arguments g, …, j in register a0 – a3
  - **Need a register for f (could be a*, s*, t*)**
  - Need to save s0-s11 on stack if it is used in this func

Leaf Procedure Example

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

RISC-V (64-bits) gcc 13.2.0

A Output... Filter... Librar

`leaf_example:`	
`addi sp,sp,-64`	adjust stack pointer to create the stack frame for the fun
`sd s0,56(sp)`	save s0 on stack
`addi s0,sp,64`	use s0 in this function
`sd a0,-40(s0)`	save a0(g) on stack -40(s0)
`sd a1,-48(s0)`	save a1(h) on stack -48(s0)
`sd a2,-56(s0)`	save a2(i) on stack -56(s0)
`sd a3,-64(s0)`	save a3(j) on stack -64(s0)
`ld a4,-40(s0)`	load g
`ld a5,-48(s0)`	load h
`add a4,a4,a5`	g+h
`ld a3,-56(s0)`	load i
`ld a5,-64(s0)`	load j
`add a5,a3,a5`	i+j
`sub a5,a4,a5`	(g+h) - (i+j)
`sd a5,-24(s0)`	store f
`ld a5,-24(s0)`	
`mv a0,a5`	return value (f) in a0
`ld s0,56(sp)`	restore s0
`addi sp,sp,64`	adjust sp back to the activation frame of the caller
`jr ra`	

Save a0 – a3 on stack, which are for arguments g, h, i, and j

Leaf Procedure Example

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

int main() {
    long long int v;
    v = leaf_example(1, 2, 3, 4);
    return v;
}
```

[See it from Compiler Explorer]

RISC-V (64-bits) gcc 13.2.0

A Output... Filter... Librarie

```
leaf_example(long long, long
        addi    sp,sp,-64
        sd      s0,56(sp)
        addi    s0,sp,64
        sd      a0,-40(s0)
        sd      a1,-48(s0)
        sd      a2,-56(s0)
        sd      a3,-64(s0)
        ld      a4,-40(s0)
        ld      a5,-48(s0)
        add     a4,a4,a5
        ld      a3,-56(s0)
        ld      a5,-64(s0)
        add     a5,a3,a5
        sub     a5,a4,a5
        sd      a5,-24(s0)
        ld      a5,-24(s0)
        mv      a0,a5
        ld      s0,56(sp)
        addi    sp,sp,64
```

```
main:
        addi    sp,sp,-32
        sd      ra,24(sp)
        sd      s0,16(sp)
        addi    s0,sp,32
        li      a3,4
        li      a2,3
        li      a1,2
        li      a0,1
        call    leaf_example
        sd      a0,-24(s0)
        ld      a5,-24(s0)
        sext.w  a5,a5
        mv      a0,a5
        ld      ra,24(sp)
        ld      s0,16(sp)
        addi    sp,sp,32
        jr      ra
```

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

```
long long int fact (long long int n){
  if (n < 1) return n;
  else return n * fact(n - 1);
}
```

- `fact` is a recursive function

See it from Compiler Explorer

```c
long long int fact (long long int n){
    if (n < 1) return n;
    else return n * fact(n - 1);
}
```

RISC-V (64-bits) gcc 13.2.0

A Output... Filter... Libraries

```
fact(long long):
        addi    sp,sp,-32
        sd      ra,24(sp)
        sd      s0,16(sp)
        addi    s0,sp,32
        sd      a0,-24(s0)
        ld      a5,-24(s0)
        bgt     a5,zero,.L2
        ld      a5,-24(s0)
        j       .L3
.L2:
        ld      a5,-24(s0)
        addi    a5,a5,-1
        mv      a0,a5
        call    fact(long long)
        mv      a4,a0
        ld      a5,-24(s0)
        mul     a5,a4,a5
.L3:
        mv      a0,a5
        ld      ra,24(sp)
        ld      s0,16(sp)
        addi    sp,sp,32
        jr      ra
```

Adjust stack frame for the call

Save return address on stack

Save s0 on stack since the func will use it

s0 now has the adjusted stack pointer

Save the argument n on stack

Load n

Branch to .L2 if n is greater than 0, i.e. reversing n<1 ➔ n>=1 ➔ n>0

If n < 1, load n to a5 for return value and jump to .L3

Load n

n – 1 is in a5

Put the argument (n-1) on a0

call fact(n-1)

move result of fact(n - 1) to a4

Load n

N * fact(n-1) and store in a5, so it is ready for return to fact(r

Return value in a5 in either both path of if, now move to a0 fo

Restore caller’s return address

Restore register s0

Pop stack

Return

Byte/Halfword/Word Operations

- RISC-V byte/halfword/word load/store
  - Load byte/halfword/word: Sign extend to 64 bits in rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - Load byte/halfword/word unsigned: Zero extend to 64 bits in rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - Store byte/halfword/word: Store rightmost 8/16/32 bits
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

String Copy Example

- C code:
  - A string is an array of characters with` \0` as the last character
    - char x[100]; a string of 100 character
    - char * x2; is used for refer to a string
  - Null-terminated string

```
void strcpy (char x[], char y[]) {
     long long int i = 0;
     while ((x[i] = y[i]) != '\0’)
        i += 1;
}
```

String Copy Example

```c
void strcpy (char x[], char y[]) {
    long long int i = 0;
    while ((x[i] = y[i]) != '\0')
        i += 1;
}
```

[See it From Compiler Explorer]

RISC-V (64-bits) gcc 13.2.0

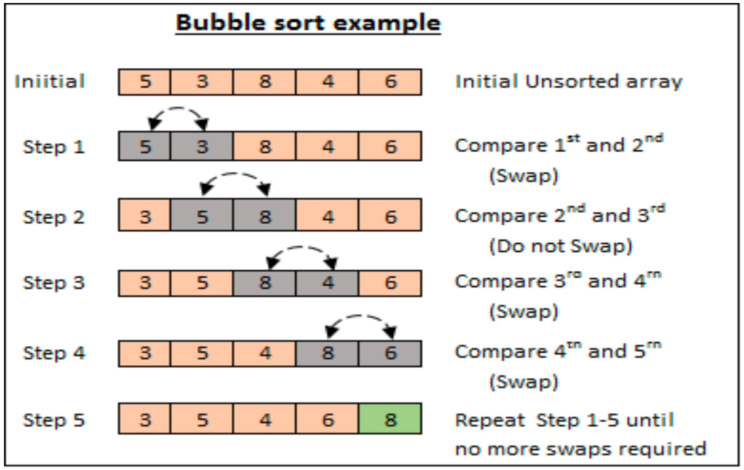
A Output... Filter... Librari

```
strcpy(char*, char*):
        addi    sp,sp,-48
        sd      s0,40(sp)
        addi    s0,sp,48
        sd      a0,-40(s0)
        sd      a1,-48(s0)
        sd      zero,-24(s0)
        j       .L2
.L3:
        ld      a5,-24(s0)
        addi    a5,a5,1
        sd      a5,-24(s0)
.L2:
        ld      a5,-24(s0)
        ld      a4,-48(s0)
        add     a4,a4,a5
        ld      a5,-24(s0)
        ld      a3,-40(s0)
        add     a5,a3,a5
        lbu     a4,0(a4)
        sb      a4,0(a5)
        lbu     a5,0(a5)
        sext.w  a5,a5
        snez    a5,a5
        andi    a5,a5,0xff
        bne     a5,zero,.L3
        nop
        nop
        ld      s0,40(sp)
        addi    sp,sp,48
        jr      ra
```

C Bubble Sort Example

```c
void swap(long long int v[], long long int k) {
    long long int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}

void sort (long long int v[], long long int n) {
    long long int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1;
             j >= 0 && v[j] > v[j + 1];
             j -= 1) {
             swap(v, j);
        }
    }
}
```



Bubble Sort Assembly From GCC

- Study from the [Code from Compiler Explorer]

```
void swap(long long int v[], long long int k) {
    long long int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}

void sort (long long int v[], long long int n) {
      long long int i, j;
      for (i = 0; i < n; i += 1) {
        for (j = i - 1;
                j >= 0 && v[j] > v[j + 1];
                j -= 1) {
          swap(v, j);
        }
      }
}
```

You Own Way of Using Register

```c
void swap(long long int v[], long long int k) {
    long long int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Register usage: v in x10, k in x11, temp in x5

```c
void sort (long long int v[], long long int n) {
    long long int i, j;
    for (i = 0; i < n; i += 1) {
      for (j = i - 1;
            j >= 0 && v[j] > v[j + 1];
            j -= 1) {
        swap(v, j);
      }
    }
}
```

Register usage: v in x10, n in x11, i in x19, j in x20

The Procedure Swap

```c
void swap(long long int v[], long long int k) {
    long long int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Register usage: v in x10, k in x11, temp in x5

```
swap:
  slli x6,x11,3     // reg x6 = k * 8
  add  x6,x10,x6    // reg x6 = v + (k * 8)
  ld   x5,0(x6)     // reg x5 (temp) = v[k]
  ld   x7,8(x6)     // reg x7 = v[k + 1]
  sd   x7,0(x6)     // v[k] = reg x7
  sd   x5,8(x6)     // v[k+1] = reg x5 (temp)
  jalr x0,0(x1)     // return to calling routine
```

The Outer Loop of Sort

```
void sort (long long int v[], long long int n) {
    long long int i, j;
    for (i = 0; i < n; i += 1) {
      for (j = i - 1;
           j >= 0 && v[j] > v[j + 1];
           j -= 1) {
        swap(v, j);
      }
    }
}
```

Register usage: v in x10, n in x11, i in x19, j in x20

- Skeleton of outer loop:
  - for (i = 0; i <n; i += 1) {

```
   mv    x21, x10        // store parameter x10 into x21
   mv    x22, x11        // store parameter x11 into x22 (not using st
   li    x19,0           // i = 0
for1tst:
   bge  x19,x11,exit1 //go to exit1 if x19≥x11(i≥n)

   (body of outer for-loop)

   addi x19,x19,1          // i += 1
   j     for1tst          // branch to test of outer loop
exit1:
```

The Inner Loop

- Skeleton of inner loop:
  - for (j = i − 1; **j >= 0 && v[j] > v[j + 1]**; j − = 1) { swap (v, j); }

```
    addi x20,x19,-1    // j = i –1
  for2tst:
    blt  x20,x0,exit2  // go to exit2 if X20 < 0 (j < 0)
    slli x5,x20,3      // reg x5 = j * 8
    add  x5,x10,x5     // reg x5 = v + (j * 8)
    ld   x6,0(x5)      // reg x6 = v[j]
    ld   x7,8(x5)      // reg x7 = v[j + 1]
    ble  x6,x7,exit2   // go to exit2 if x6 ≤ x7
    mv   x10, x21      // first swap parameter is v
    mv   x11, x20      // second swap parameter is j
    jal  x1,swap    // call swap
    addi x20,x20,-1    // j -= 1
    j    for2tst    // branch to test of inner loop
 exit2:
```

Preserving Registers

- Preserve saved registers:

```
addi sp,sp,-40  // make room on stack for 5 regs
sd   x1,32(sp)  // save x1 on stack
sd   x22,24(sp) // save x22 on stack
sd   x21,16(sp) // save x21 on stack
sd   x20,8(sp)  // save x20 on stack
sd   x19,0(sp)  // save x19 on stack
```

- Restore saved registers:

```
exit1:
  ld   x19,0(sp)  // restore x19 from stack
  ld   x20,8(sp)  // restore x20 from stack
  ld   x21,16(sp) // restore x21 from stack
  ld   x22,24(sp) // restore x22 from stack
  ld   x1,32(sp)  // restore x1 from stack
  addi sp,sp, 40  // restore stack pointer
  jalr x0,0(x1)
```

The Full Version

Saving registers		
	`sort:` `addi sp, sp, -40`	`# make room on stack for 5 registers`
	`sd x1, 32(sp)`	`# save return address on stack`
	`sd x22, 24(sp)`	`# save x22 on stack`
	`sd x21, 16(sp)`	`# save x21 on stack`
	`sd x20, 8(sp)`	`# save x20 on stack`
	`sd x19, 0(sp)`	`# save x19 on stack`

Procedure body		
Move parameters	`mv x21, x10`	`# copy parameter x10 into x21`
	`mv x22, x11`	`# copy parameter x11 into x22`
Outer loop	`li x19, 0`	`# i = 0`
	`for1tst:bge x19, x22, exit1`	`# go to exit1 if i >= n`
Inner loop	`addi x20, x19, -1`	`# j = i - 1`
	`for2tst:blt x20, x0, exit2`	`# go to exit2 if j < 0`
	`slli x5, x20, 3`	`# x5 = j * 8`
	`add x5, x21, x5`	`# x5 = v + (j * 8)`
	`ld x6, 0(x5)`	`# x6 = v[j]`
	`ld x7, 8(x5)`	`# x7 = v[j + 1]`
	`ble x6, x7, exit2`	`# go to exit2 if x6 < x7`
Pass parameters and call	`mv x10, x21`	`# first swap parameter is v`
	`mv x11, x20`	`# second swap parameter is j`
	`jal x1, swap`	`# call swap`
Inner loop	`addi x20, x20, -1`	`j for2tst`
	`j for2tst`	`# go to for2tst`
Outer loop	`exit2:` `addi x19, x19, 1`	`# i += 1`
	`j for1tst`	`# go to for1tst`

Restoring registers		
	`exit1:` `ld x19, 0(sp)`	`# restore x19 from stack`
	`ld x20, 8(sp)`	`# restore x20 from stack`
	`ld x21, 16(sp)`	`# restore x21 from stack`
	`ld x22, 24(sp)`	`# restore x22 from stack`
	`ld x1, 32(sp)`	`# restore return address from stack`
	`addi sp, sp, 40`	`# restore stack pointer`

Procedure return		
	`jalr x0, 0(x1)`	`# return to calling routine`

RISC-V Instruction Set Extensions

- M: integer multiply, divide, remainder
- A: atomic memory operations
- F: single-precision floating point
- D: double-precision floating point
- C: compressed instructions
  - 16-bit encoding for frequently used instructions

The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

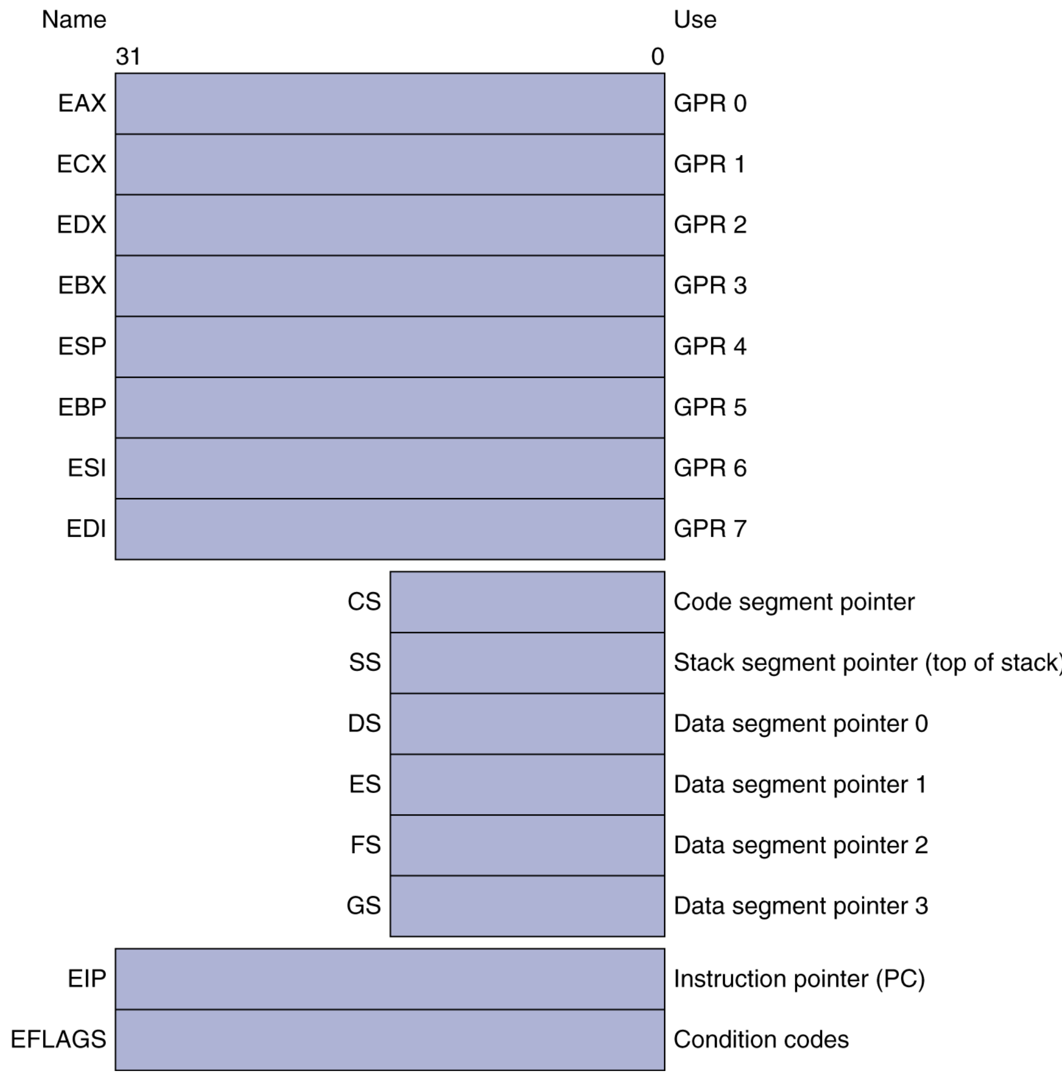
The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

Basic x86 Registers



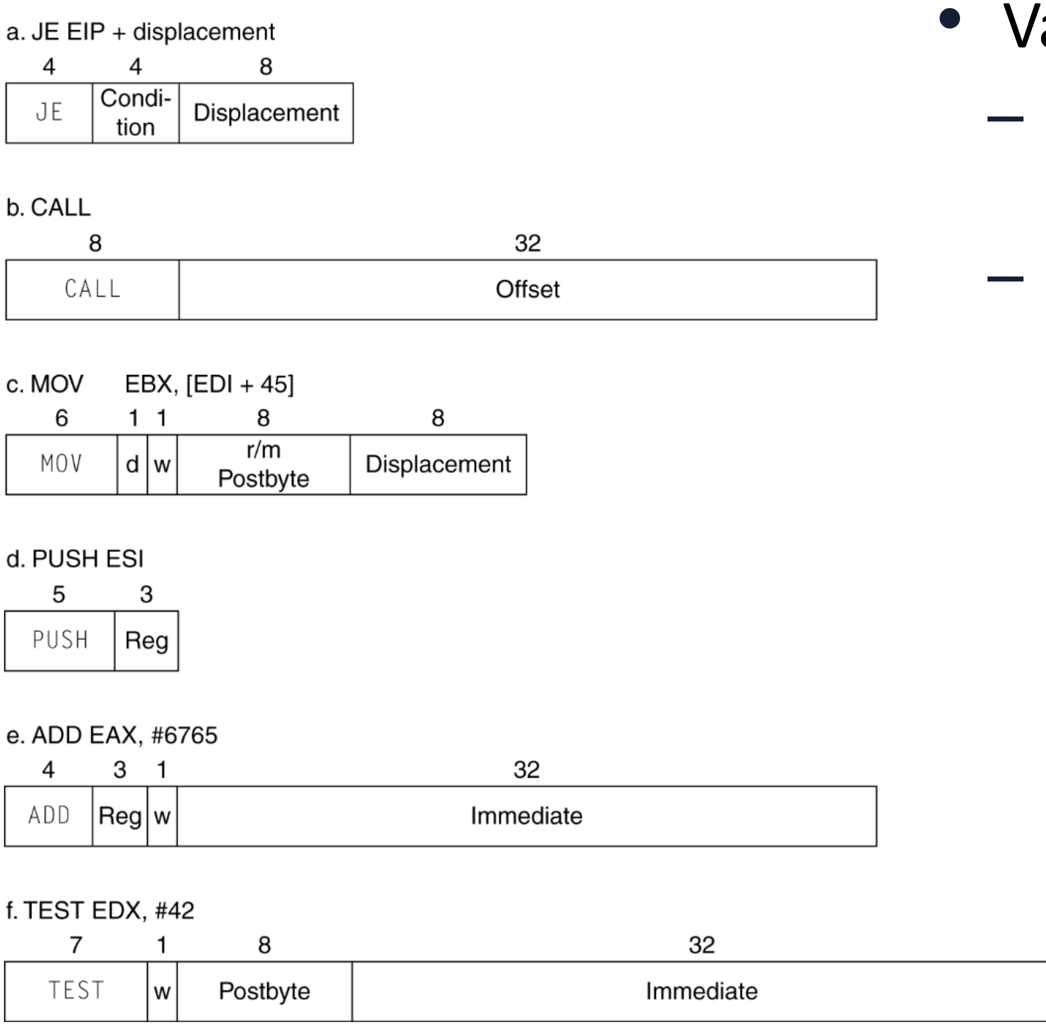
Basic x86 Addressing Modes

- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding



- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

More Materials for RISC-V Instruction

- Slides for RISC-V intro and specification:
  - https://passlab.github.io/ITSC3181/notes/lectureXX_RISCV_ISA.pdf
- RISC-V instruction reference cards:
  - https://passlab.github.io/ITSC3181/resources/RISCVGreenCardv8-20151013.pdf
- Information for learning assembly programming
  - https://passlab.github.io/ITSC3181/resources/RISC-VAssemblyProgramming.html
- Resources from the official website including the standard
  - https://riscv.org/

Concluding Remarks

- Instruction Set Architecture are Hardware and Software Interface
- Three major classes of instructions
  - Arithmetic and logic instructions
  - Load/Store instructions
  - Control transfer (branch and jump/link)
  - Other helpful instruction, e.g. load immediate, etc.
- High-level language constructs to instruction sequence
  - Arithmetic and logic expression => Arithmetic and logic instructions
  - Array reference => address calculation and load/store
  - If-else/switch-case, for/while-loop => branch and jump
  - Function call => jump/link, store and restore registers

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Good design demands good compromises
  4. Make the common case fast