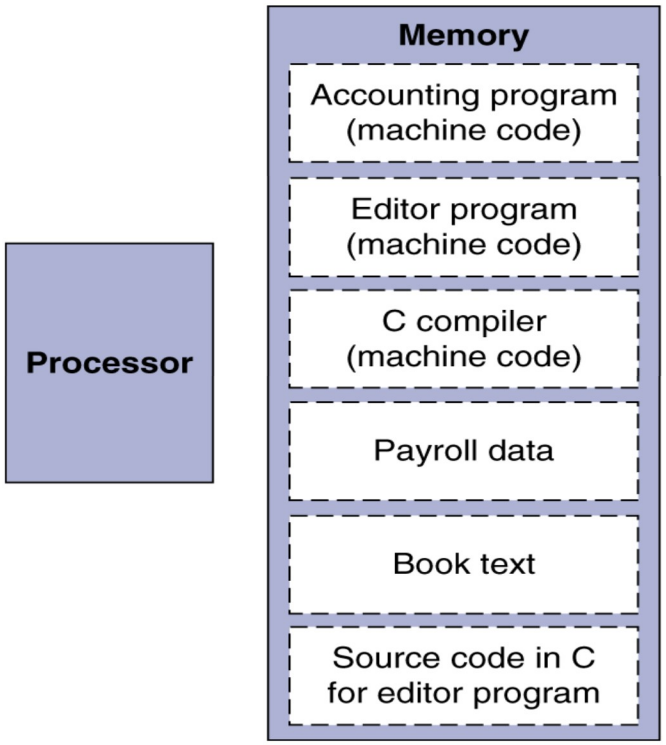
# Topics for Chapter 2

- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS
  - Memory access instructions (Chapters 2.3)
  - Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 2:**
  - ➔ Procedure Calls (Chapters 2.8)
- **Part 3:**
  - RISC-V Instruction Format (Chapter 2.5)
  - RISC-V Addressing Modes (Chapter 2.10)

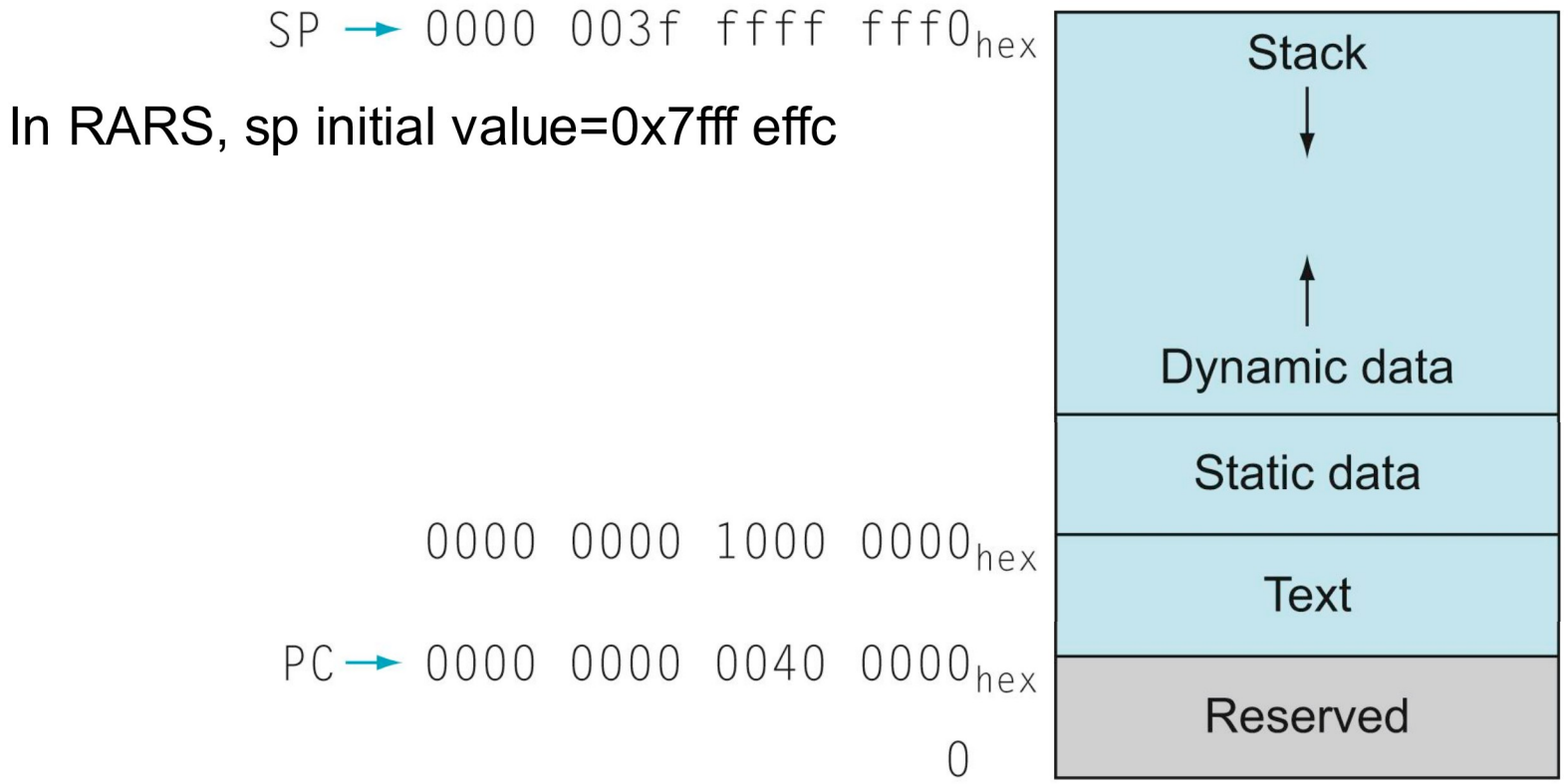
# Stored Program Concept

**The BIG Picture**




- Instructions and data are stored in memory.
- Each register stores how many bits? how many bytes? how many words?
- How much data can I access from memory at a time?
- Each instruction is how many bits?
- Memory addresses are how many bits?
- RISC-V memory capacity in bytes? in words?

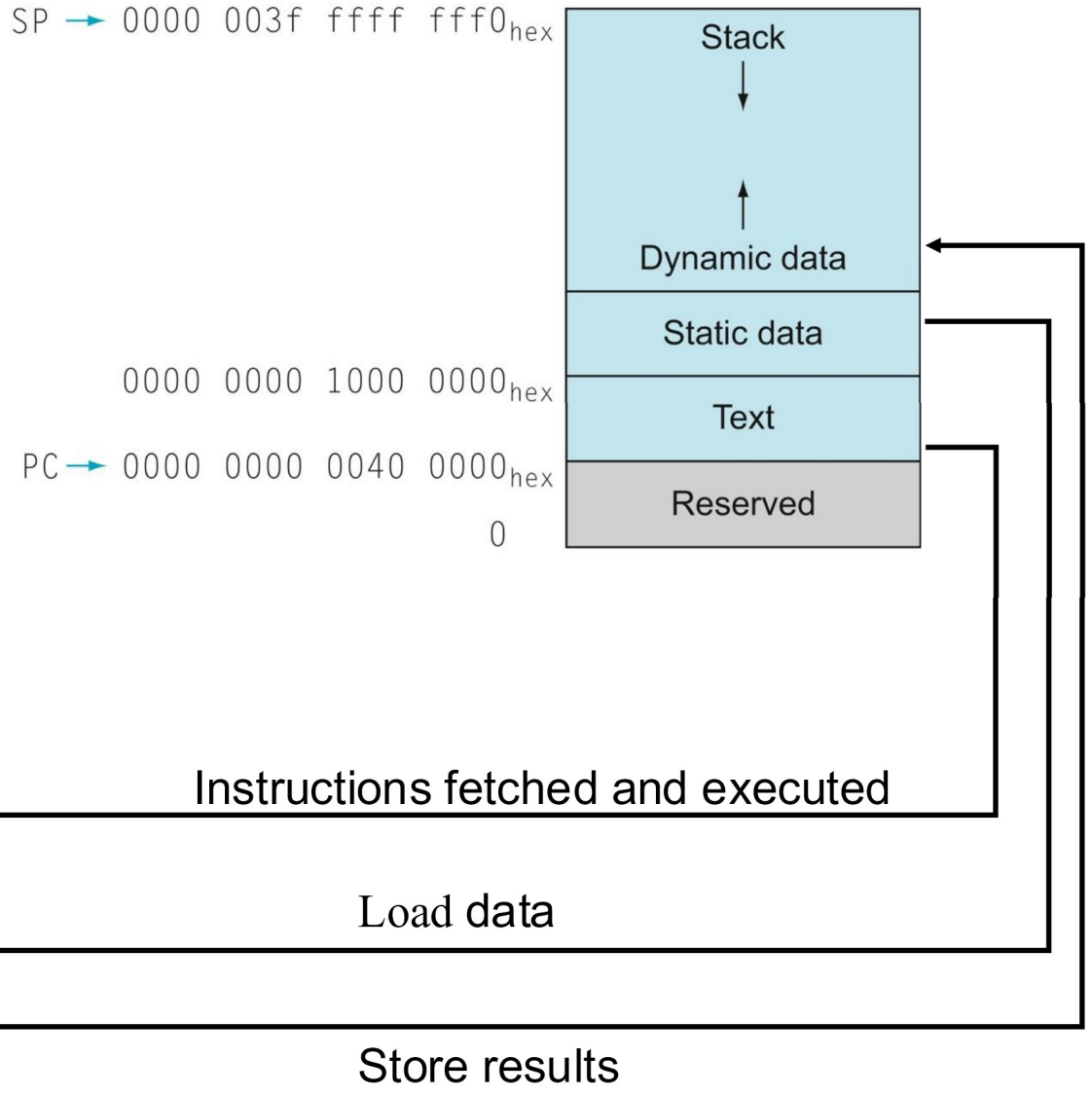
# RISC-V Memory Allocation




- **Static: Variables declared once per program, cease to exist after execution completes. e.g., C globals, arrays**
- **Heap: Variables declared dynamically (malloc)**
- **Stack: Space to be used by procedure during execution; this is where we can save register values**

# RISC-V Memory Allocation

- **Text segment**
  - Program code
  - Addresses 0x0040 0000 to 0x0FFF FFFF
- **Data segment**
  - Addresses 0x1000 0000 to 0xFFFF FFFF



# The Program Counter (PC)

- **Special purpose register**
- **Not available to programmer**
- **Holds address of?**
- **Assuming currently executing the first instruction**
  - $(5048)_{10}$ = 0x13B8
  - PC = 0x0000 _____
  - PC + 4 = 0x0000 _____
  - Byte stored at 0x0000 13BA?

This is the machine representation of:
`addi s0, x0, 0`

| Memory Location | Instruction |
|---|---|
| 5048 | 0X00000413 |
| 5052 | addi s1, x0, 1 |
| 5056 | beq s1, s0, continue |
| 5060 | jal x0, exit |
| 5064 | |
| 5068 (continue) | |
| … | |
| 6000(exit) | |

# Procedure Call in C Program

```
main() {
  int i,j,k,m;
  ...
  i = mult(j,k); ...
  m = mult(i,i); ...
}
/* really dumb mult function
  */
int mult (int mcand, int
  mlier){
  int product;
  product = 0;
  while (mlier > 0)  {
   product = product + mcand;
   mlier = mlier -1; }
  return product;
}
```

**What information must a compiler/programmer keep track of?**

**What instructions can accomplish this?**

# Procedure Call Bookkeeping

- **Use registers**
- **Register conventions:**
  - Return address ra
  - Arguments a0, a1, a2, …, a7
  - Return value a0, a1
  - Local variables s0, s1, … , s11
  - Temporary variables t0, t1, …, t6
- **More variables?**
  - Spill registers: use the stack in memory

# RISC-V Support for Function Calls

C

```
... sum(a,b);... /* a,b:s0,s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

RISC-V

```
address
1000 add  a0,s0,zero   # x = a
1004 add  a1,s1,zero   # y = b
1008 jal ra, sum         # new instruction
1012 ...

2000 sum: add a0,a0,a1
2004 jalr  zero, ra,0    # new instruction
```

# RISC-V Instruction: jal

- **Syntax for `jal` (jump and link)**
  - jal ra, label
- **`jal` should really be called `laj` for "link and jump":**
  - Step 1 (link): Save address of *next* instruction (i.e., PC+4) into `ra`
    - Why next instruction? Why not current one?
  - Step 2 (jump): Jump to the given label
- **Why jal?**
  - ra automatically saves PC+4
  - No need to know where the code is loaded into memory
  - Make the common case (function calls) fast

# RISC-V Instruction: jalr

- **Syntax for `jalr` (jump register):**

  jalr saved_addr, jump_addr, imm

  - saved_addr: where curr instruction address+4 will be stored
  - jump_addr: contains address to jump to
  - imm: added to jump_addr (always 0 for our needs)
- **Why use `jalr`?**
  - a function might be called many times, so we can't return to a fixed place.

# *Use of jalr*

- **ret and jr pseudo-instructions**

  jalr x0, ra, 0

- **Call function at any 32-bit absolute address**

  lui x1, <hi20bits>

  jalr ra, x1, <lo12bits>

- **Jump PC-relative with 32-bit offset**

  auipc x1, <hi20bits>

  jalr x0, x1, <lo12bits>

# RISC-V Instructions for Function Calls

| | | | | | | |
|---|---|---|---|---|---|---|
| `jal` | Jump And Link | J | `1101111` | | | `rd = PC+4; PC += imm` |
| `jalr` | Jump And Link Reg | I | `1100111` | `0x0` | | `rd = PC+4; PC = rs1 + imm` |

# Steps for Making a Procedure Call (V. 1)

1) Place parameter in registers → a0-a7

2) Call procedure → jal ra, PROC

**ra is x1**

3) Perform procedure's operations

4) Place result in register for caller → a0, a1

5) return to caller → jalr x0, ra, 0

# Caller

# Callee

`jal ra, ProcedureLabel`

`jalr zero, ra, 0`

| Address | Instruction |
|---|---|
| 4060 | `add ...` |
| 4064 | `ori ...` |
| 4068 | `and ...` |
| 4072 | `jal ra, ProcedureLabel` |
| 4076 | `sll ...` |

ra = 4076

| Address | Instruction |
|---|---|
| 5044 ProcedureLabel: | `add...` |

| Address | Instruction |
|---|---|
| 5060 | `srl...` |
| 5064 | `jalr zero, ra, 0` |

| Address | Instruction |
|---|---|
| 4076 | `sll ...` |

- `jal ra, LABEL` stores PC+4 in `ra`
  - Used by caller
- `jalr x0, ra, 0` uses `ra` to modify PC
  - Used by callee

# Procedure Call Example: C Code

```
main() {
  int i,j,k,m; /* i-m:s0-s3 */
  ...
  i = mult(j,k); ...
  m = mult(i,i); ...
}
int mult (int mcand, int mlier){
  int product;
 product = 0;
  while (mlier > 0)  {
   product += mcand;
   mlier -= 1; }
  return product;
  }
```

What are the arguments?

What are the results?

# Procedure Call Example: main function

```
main:
  addi a0,s1,0          # arg0 = j
  addi a1,s2,0          # arg1 = k
  jal ra, mult          # call mult
  addi s0,a0,0          # i = mult()
  ...
# arg0 = i, already in a0
  addi a1,s0,0          # arg1 = i
  jal ra, mult          # call mult
  addi s3,a0,0          # m = mult()
  ...
  addi a7, zero,10
  ecall
```

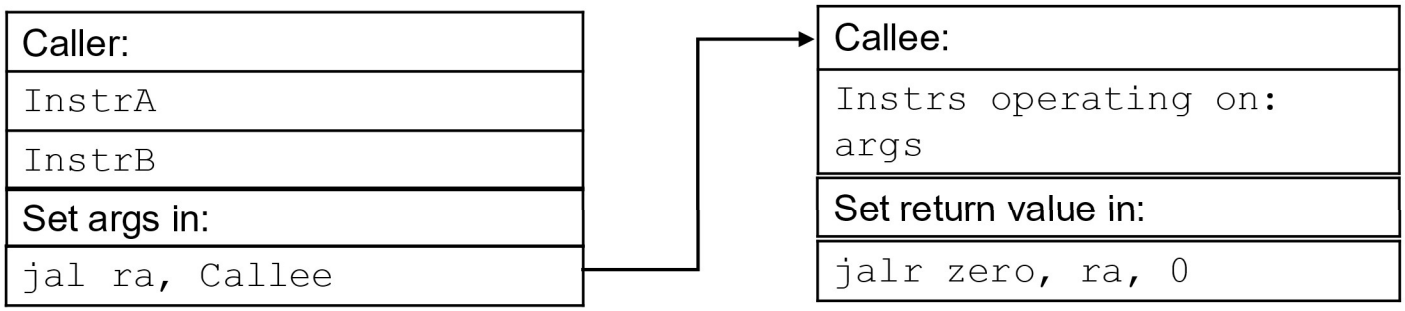
```
main() {
int i,j,k,m; /* i-m:s0-s3 */
...
i = mult(j,k); ...
m = mult(i,i); ... }
```

# Procedure Call Example: sub function

```
mult:
      addi  t0,zero,0          # prod=0
Loop:
     bge  0,a1,Fin        # if mlr <= 0, goto Fin
     add  t0,t0,a0        # prod+=mc
     addi a1,a1,-1        # mlr-=1
     jal zero,Loop         # goto Loop
Fin:
     addi  a0,t0,0        # a0=prod
     jalr zero, ra, 0     # return
```

```
int mult (int mcand, int mlier){
int product = 0;
while (mlier > 0)  {
 product += mcand;
 mlier -= 1; }
return product;
}
```

# What Do We Know So Far?




- In RISC-V, "argument" registers for passing parameters are?
- jal instruction does two actions, those are?
- RISC-V procedure returns values in?
- jalr zero, ra, 0 jumps to address in ra, which is?