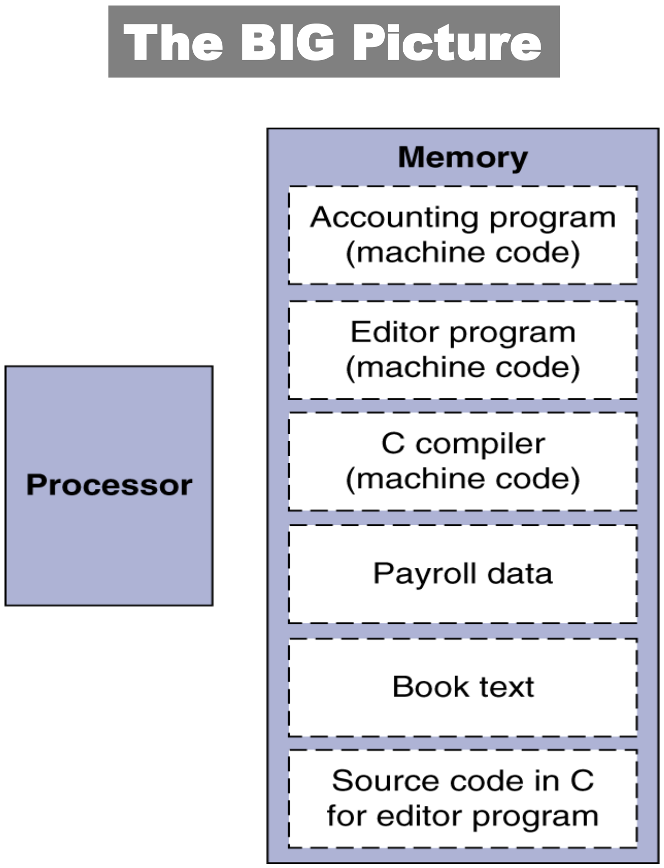
Topics for Chapter 2

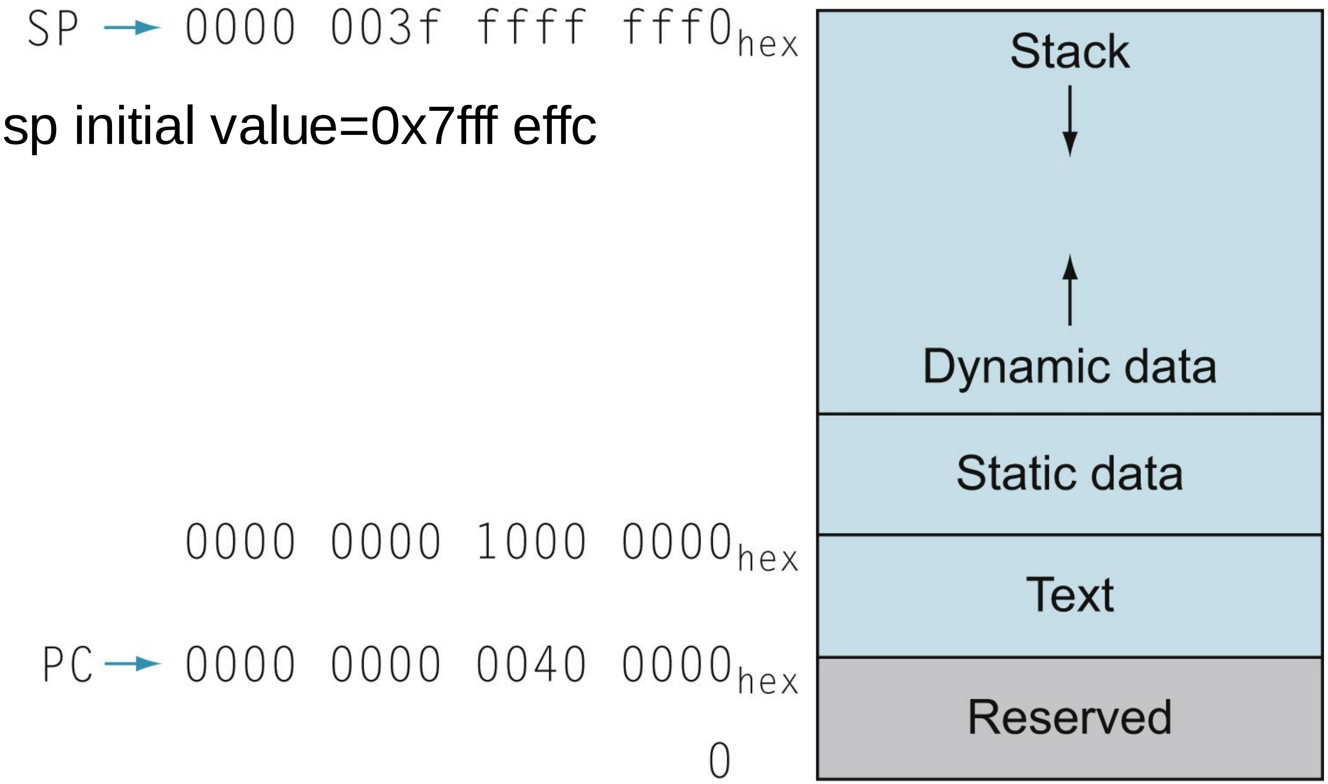
- **Part 1**
  - RISC-V Instruction Set (Chapter 2.1)
  - Arithmetic instructions (Chapter 2.2)
  - Introduction to RARS
  - Memory access instructions (Chapters 2.3)
  - Bitwise instructions (Chapter 2.6)
  - Decision making instructions (Chapter 2.7)
  - Arrays vs. pointers (Chapter 2.14)
- **Part 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



- 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

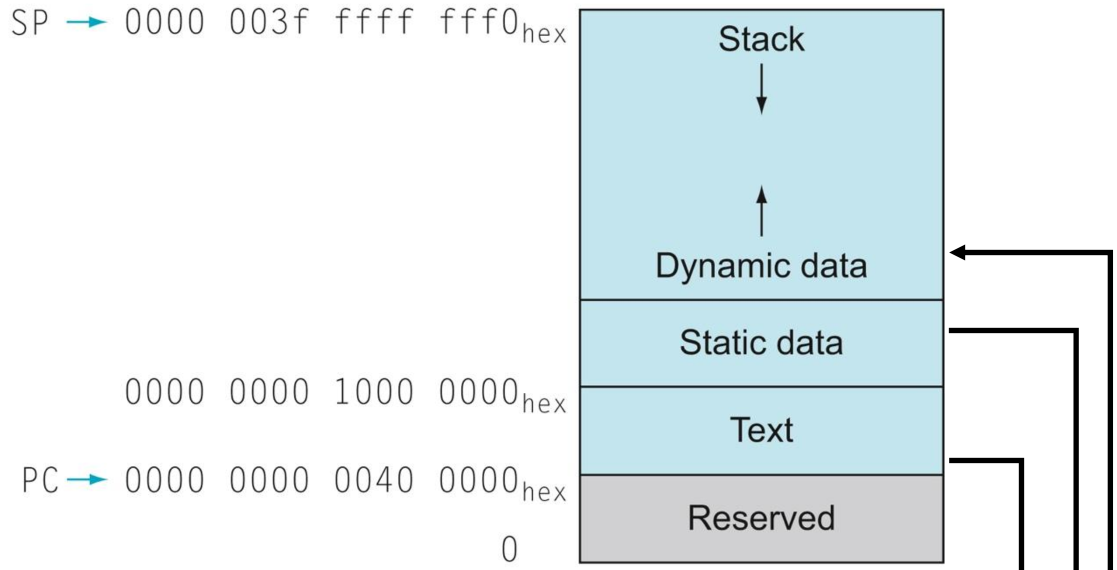


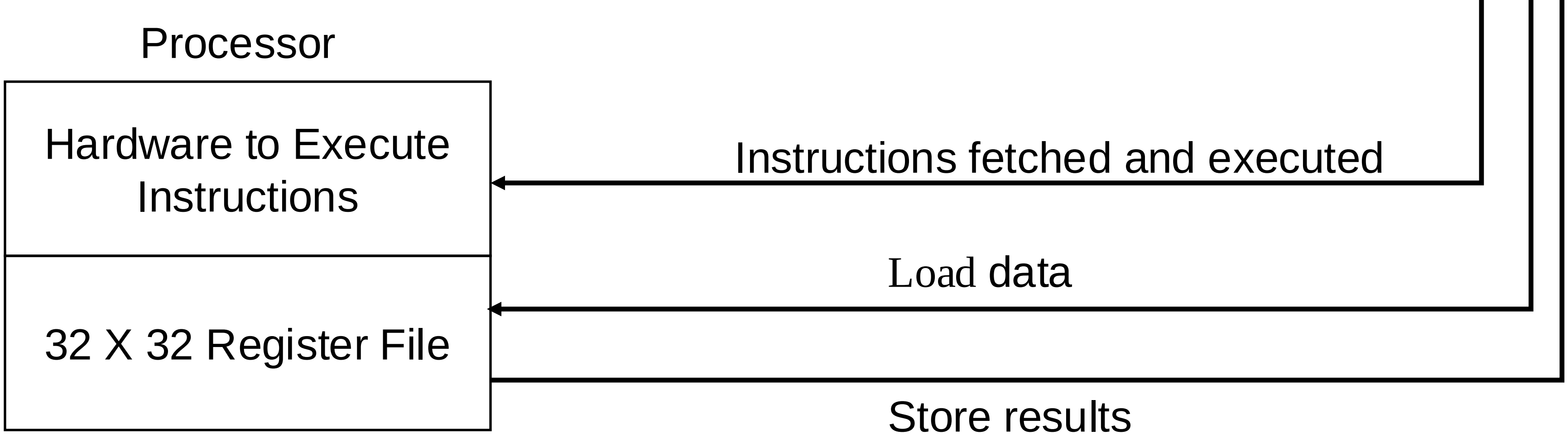
In RARS, sp initial value=0x7fff effc

- **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)

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

- **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?

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
```

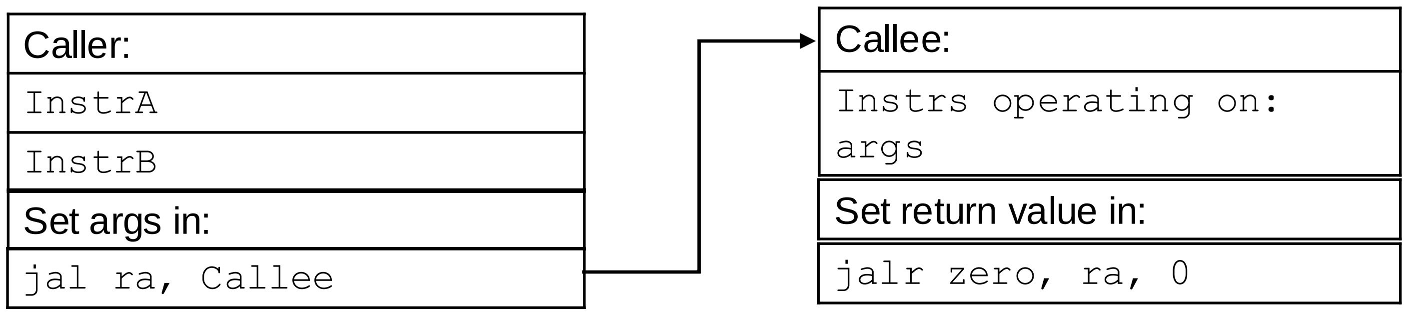
```
main() {
int i,j,k,m; /* i-m:s0-s3 */
...
i = mult(j,k); ...
m = mult(i,i); ... }
```

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?

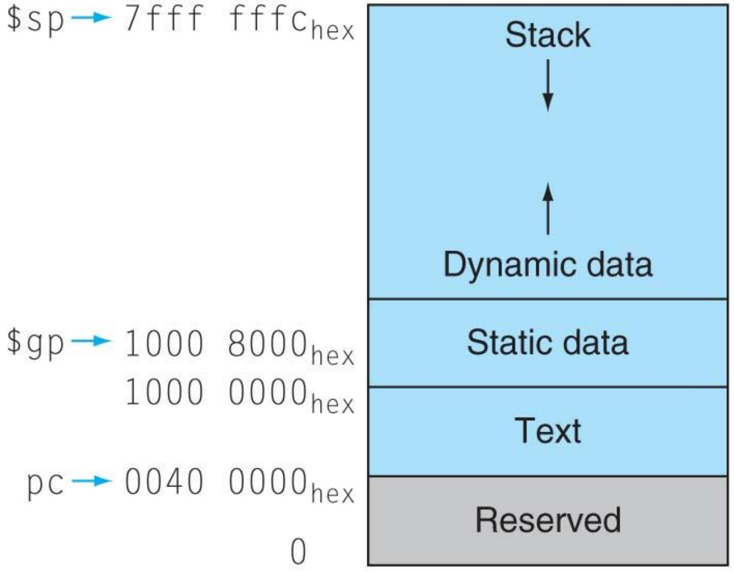
Nested Procedures

```
int sumSquare(int x, int y) {
     return mult(x,x)+ y;
 }
```

- **Something called `sumSquare`, now `sumSquare` is calling `mult`.**
- **`ra` may be overwritten**
  - Need to save `sumSquare` return address before call to `mult`.
- **In general, may need to save some other info in addition to `ra`.**

Using the Stack to Store Other Info

- **register `sp` : the last used space in the stack.**
- **When a0-a7 are not enough**
  - Spill arguments to the stack
- **If the caller uses s0 before and after the procedure call and the callee also uses s0**
  - Preserve registers in stack
- **Stack grows downwards**



Stack Pointer (sp)

- **Available to programmer**
- **Allocate space on stack**
  - Decrement sp
  - Grow from top down
- **Use space on stack**
  - Push: sw reg, offset (sp)
  - Pop: lw reg, offset (sp)
- **Release space on stack**
  - Increment sp
  - Shrink towards top

```
square_sum:
    addi sp, sp, -4
    sw ra, 0(sp)

    ...

    lw ra, 0(sp)
    addi sp, sp, 4
    jalr zero, ra, 0
```

Popping does not make the data disappear.
Neither does releasing the space!

Steps for Making a Procedure Call (V. 2)

1) Caller saves necessary values onto stack.
2) Assign argument(s) in a0-a7, if any.
3) `jal` ra, proc -- call procedure
4) Perform procedure's operations
5) Place result in a0, a1 if any, for caller
6) jalr x0, ra, 0 – return to caller
7) Caller restore values from stack.

Using the Stack: Example

```
int sumSquare(int x, int y) {
   return mult(x,x)+ y; }
```

```
sumSquare:
        addi sp,sp,-8   # space on stack
"push"  sw ra, 4(sp)       # save ret addr
        sw a1, 0(sp)       # save y

        add a1,a0,zero # mult(x,x)
        jal ra, mult       # call mult

        lw a1, 0(sp)      # restore y
        add a0,a0,a1    # mult()+y
        lw ra, 4(sp)      # get ret addr
"pop"   addi sp,sp,8    # restore stack
        jalr zero, ra, 0
mult: ...
```

Rules for Procedures

- **Called with a `jal` instruction, returns with a `jalr zero,ra,0`**
- **Accepts up to 7 arguments in `a0`, `a1`, `a2`, …, `a7`**
- **Return value is always in `a0` (and if necessary in `a1`)**
- **Must follow register conventions (even in functions that only you will call)!**
  - A set of generally accepted rules as to which registers will be unchanged after a procedure call (`jal`) and which may be changed.

Register Conventions

- **Calle<u>R</u>: the calling function**

- **Calle<u>E</u>: the function being called**

- **When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.**

Caller vs. Callee

- **Caller has to save if it wants to preserve the value around a function call**
  - a0-a1 (can’t put back in a0/a1 if return values use it)
  - a2-a7
  - t0-t6
  - ra: only needs to save once no mater how many calls it makes
- **Callee has to save if it wants to modify the value in its body**
  - s0 – s11
  - sp: handles this one differently – release what you allocate
  - gp: we will not use this, so not saved
  - tp: we will not use this, so not saved

Where to Save These Registers

- **The stack**
- **The code is in complete control of the stack pointer**
  - Allocate by
  - Release by
- **How does it know how much space it needs?**

How to Save sp?

- **Procedure "knows" how much it changed sp by at the start**
- **Assumes every procedure it calls respects this also**
- **Procedure "restores" sp by restoring the change**
- **Don't store sp on the stack**

- **If procedure changed sp and fails to restore sp, caller can't locate its own stack items**
- **Recommend: only change sp at procedure entry and exit**

RISC-V Registers

REGISTER NAME, USE, CALLING CONVENTION

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

CSCI 341 Simplifications

- **Assume no one calls main**
  - Main is only a caller, never a callee
- **gp, tp: don’t worry about them**
  - You can write perfectly good RISC-V code without them.

Registers- saved/preserved

- **s0-s11: Restore if callee changes.**
  - that's why they're called saved registers.
  - If the callee changes these in any way, it must restore the original values before returning.
- **sp: Restore if callee changes.**
  - Called must ensure it point to the same place before and after the jal call, so the caller can restore values from the stack.
- **HINT -- All saved registers start with s!**
- **Don't put sp on the stack**

Registers- volatile

- **ra**
  - The jal call itself will change this register.
  - Caller needs to save on stack if nested call.
- **a0-a1**
  - contain the new returned values.
- **a2-a7**
  - volatile argument registers.
  - Caller needs to save if need old values after the call.
- **t0-t6**
  - That’s why they’re called temporary
  - any procedure may change them at any time
  - Caller needs to save if they’ll need same values after the call

Implications of Register Conventions

- **If function R calls function E,**
  - R must save any temporary registers that it may be using onto the stack before making a `jal` call.
  - E must save any S (saved) registers it intends to use before garbling up their values
- **Remember: Caller/callee need to save only temporary/saved registers they are using, not all registers.**

Steps for Making a Procedure Call (V.3, final version)

1) Caller saves needed volatile regs onto stack.
2) Assign argument(s) in a0-a7, if any.
3) `jal` ra, proc -- call procedure
4) Callee allocates stack and preserves used s-regs
5) Perform procedure’s operations
6) Callee releases stack and restores used s-regs
7) Place result in a0, a1 if any, for caller
8) jalr x0, ra, 0 – return to caller
9) Caller restore values from stack.

Procedure Call Conventions (Rules, to you)

- **Call with** jal ra, proc**; return with** jalr x0, ra, 0
  - Maintain the "fiction" of procedures with jal/jalr pairs
  - No other jumps between two procedure bodies, only within a procedure
- **Accept up to 8 arguments in a0-a7**
  - Always used in that order (can't skip or get clever to avoid stack use)
  - We will never a 9^{th} argument (simplifying assumption)
- **Return value is always in a0 (and a1 if two return values)**
- **Follow register conventions**
  - Even in function that only you call!
  - These dictate which registers will be unchanged after a procedure call and which may be changed
  - Assume the worst: that they are changed, every time!

Nested Procedure Call Example (C Code)

```
main() {
  int i,j,k; /* i-k:s0-s2 */
  ...
  k = sumSquare(i,j); ...
  }

int sumSquare(int x, int y) {
      return mult(x,x)+ y;
  }

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

RISC-V Code for main

```
main:
  addi a0,s0,0          # arg0 = i
  addi a1,s1,0          # arg1 = j
  jal ra, sumSquare     # call sumSquare
  addi s2,a0,0          # k = mult()

  addi a7, x0, 10
  ecall
```

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

k = sumSquare(i,j);
}
```

Comments for main

- **main function ends with** ecall 10**, not** jalr x0,ra,0**, there is no need to save ra onto stack**
- **all variables used in main function are saved registers, so there's no need to save these onto stack**
- **main cannot rely on procedures not touching and changing any volatile registers – this includes a0-a7**

RISC-V Code for sumSquare

```
int sumSquare(int x, int y) {
   return mult(x,x)+ y; }
```

```
sumSquare:
        addi sp,sp,-8   # space on stack
        sw ra, 4(sp)       # save ret addr
        sw a1, 0(sp)       # save y


        add a1,a0,zero # mult(x,x)
        jal ra, mult         # call mult

        lw a1, 0(sp)      # restore y
        add a0,a0,a1     # mult()+y
        lw ra, 4(sp)      # get ret addr
        addi sp,sp,8     # restore stack
        jalr zero, ra, 0
```

sumSquare: Caller and Callee

- **Has jal, so has to save/restore ra, and thus sp**
- **Only known values coming in are sp, ra, a0, a1**
- **No good choice for saved regs vs. temp regs. , can't know what caller used vs. what callee used – so stick with our original conventions**
  - s for local variables
  - t for intermediate results

RISC-V code for mult (leaf procedure)

```
mult:
     addi   t0,x0,0          # 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;
}
```

Comments for leaf procedure

- `mult` **is a leaf procedure!**
  - No `jal` calls are made from `mult`
- **No need to save/restore ra since it won't change**
- **Only known values coming in are ra, a0, a1**
- **Use volatile registers (t0-t6, also include a0-a7) for local variables in leaf procedures!**
- **if it uses saved registers, it must save/restore them – so avoid that, to avoid having to use the stack**
- **Why use temp registers not s registers for intermediate calculations?**
  - Since you are the compiler, bend the rules (t registers not just for intermediate results in a leaf)

Review Procedures So Far

1) **Consider a nested procedure P that calls procedure Q. P should always save ra. True or False?**

2) **If P needs t0, and will write a0 to pass a parameter to Q, P might first**

a) need to save a0 to the stack.	b) need to return to the main program	c) need to save t0 to the stack.

3) **When P calls Q, P should expect**

a) Q might pop less from the stack than Q pushed to the stack	b) Q will push to stack above the stack pointer	c) Q will pop from stack as much as Q pushed to the stack

Recursion: Caller IS Callee

- **Each call creates a new stack frame**
- **special case of nested procedure call**

Classic Factorial Problem in RISC-V

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

- Argument n in a0
- Result in a0

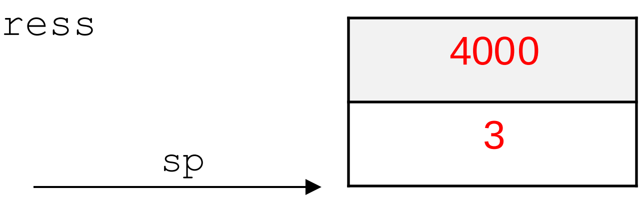
fact:	# Procedure Address 8000	
addi sp, sp, -8	# adjust stack for 2 items	Instructions 1 – 3
sw ra, 4(sp)	# save the return address	
sw a0, 0(sp)	# save the argument n	
addi t0, zero, 1	# t0 = 1	Instructions 4 – 5
blt t0, a0, L1	# if n > 1, go to L1	
addi a0, zero, 1	# return 1 (n<=1)	Instructions 6 – 8
addi sp, sp, 8	# restore stack pointer	
jalr zero, ra, 0	# return to caller	
L1: addi a0, a0, -1	# n >= 1: argument gets (n - 1)	Instructions 9 – 10
jal ra, fact	# call fact with (n -1)	
lw t0, 0(sp)	# return from jal: restore n to t0	Instructions 11 – 13
lw ra, 4(sp)	# restore the return address	
addi sp, sp, 8	# adjust stack pointer: pop 2 items	
mul a0, t0, a0	# return n * fact (n - 1)	Instructions 14 – 15
jalr zero, ra, 0	# return to the caller	

`Main()` calls `fact()`: First Call (n = 3)

- **Assume `jal ra, fact` is located at location 3996 in `main()`**
- **Argument n = 3 in `a0`**
- **Return address for `main()` = ________ so `ra` = ______**

- Instructions 1 – 3.
- Fill out the stack after the execution of the following instructions
- `sp` already adjusted in picture

```
addi sp, sp, -8      # adjust stack for 2 items
sw   ra, 4(sp)       # save return address
sw   a0, 0(sp)       # save argument
```



Instructions 4 – 5

- **Test for ___a0 > 1___**
- **a0 = _____**

will blt jump or continue?

```
addi t0, zero, 1                    # test for n > 1
blt  t0, a0, L1
```

Instructions 9 – 10

- **Jump to L1**
- **After executing instructions 9 and 10**

```
L1: addi a0, a0, -1      # n > 1: argument gets (n - 1)
    jal  ra, fact                # call fact with (n -1)
```

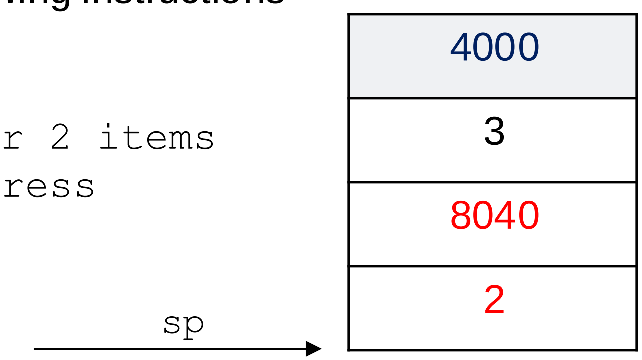
- a0 updated to 2
- address of fact = 8000
- `ra` = 8040

`fact()` calls `fact()`: Second Call (n = 2)

- **Argument n = 2 in a0**
- **Return address for `fact()` = ________, so `ra` = ______**

  - Instructions 1 – 3.
  - Fill out the stack after the execution of the following instructions
  - `sp` already adjusted

```
addi sp, sp, -8      # adjust stack for 2 items
sw   ra, 4(sp)       # save return address
sw   a0, 0(sp)       # save argument
```



Instructions 4 – 5

- **Test for** ____$a0 > 1$______
- **a0 =** ____

will blt jump or continue?

```
addi t0, zero, 1                    # test for n > 1
blt  t0, a0, L1
```

Instructions 9 – 10

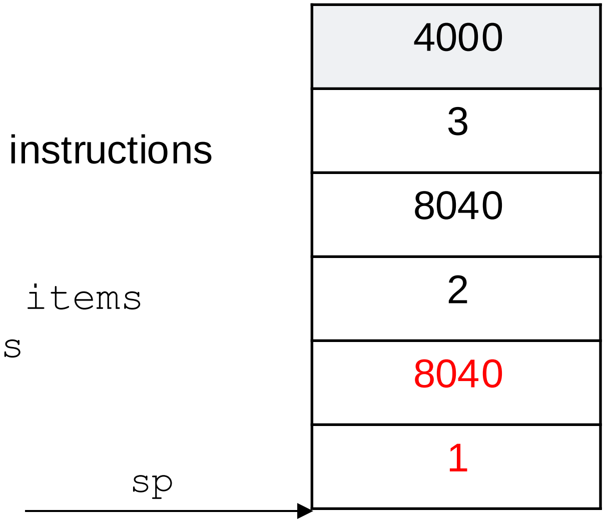
- **Jump to L1**
- **After executing instructions 9 and 10**

```
L1: addi a0,a0,-1        # n > 1: argument gets (n - 1)
    jal ra, fact         # call fact with (n -1)
```

- a0 updated to 1
- address of fact = 8000
- `ra` = ____8040____

`fact()` calls `fact()`: Third Call (n = 1)

- **Argument n = 1 in a0**
- **Return address for `fact()` = ________ so (`ra` = _____)**



- Instructions 1 – 3.
- Fill out the stack after the execution of the following instructions
- `sp` already adjusted

```
addi sp, sp, -8      # adjust stack for 2 items
sw   ra, 4(sp)       # save return address
sw   a0, 0(sp)       # save argument
```

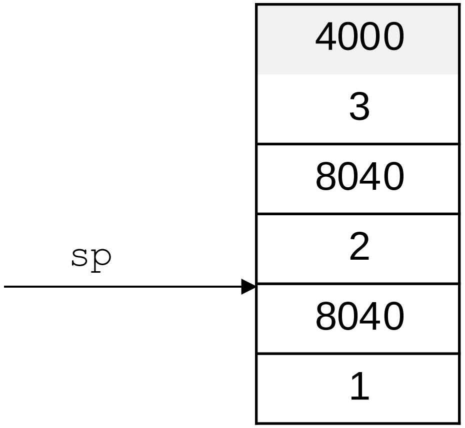
Instructions 4 – 5

- **Test for** $\underline{\quad a0 > 1 \quad\quad\quad}$
- **a0 = ____**

  will blt jump or continue?

```
addi t0, zero, 1                # test for n > 1
blt  t0, a0, L1
```

Instructions 6 – 8



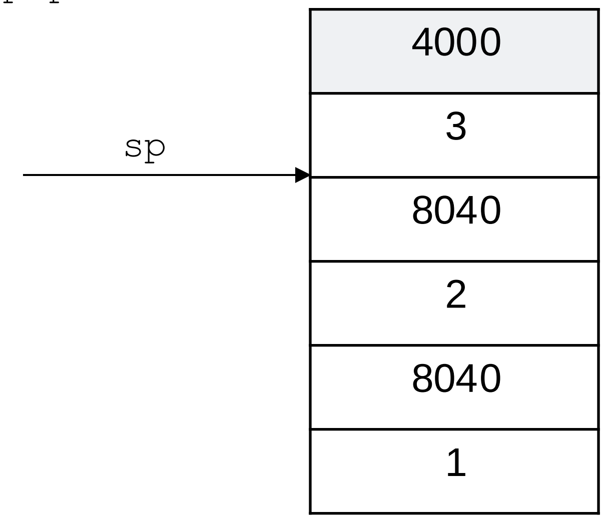

```
addi a0, zero, 1 # return 1
addi sp, sp, 8   # restore stack pointer
jalr zero, ra, 0 # return to caller
```

- **Return to** ___8040___ **with a0 =** ___1___
- **New PC =** ___8040___
- **How many times are these 3 instructions executed over the duration of the program (until the completion of the calculation of the factorial)?**
- **We adjusted `sp` but never popped anything off. Why?**

Instructions 11 – 13

```
lw t0, 0(sp)      # return from jal: recover argument n
lw ra, 4(sp)      # restore the return address
addi sp, sp, 8    # adjust stack pointer to pop 2 items
```

- **a0 =1 (from returned call)**
- **t0 = 2**
- **ra = 8040**
- **What is left in the stack after the execution of the instructions above?**



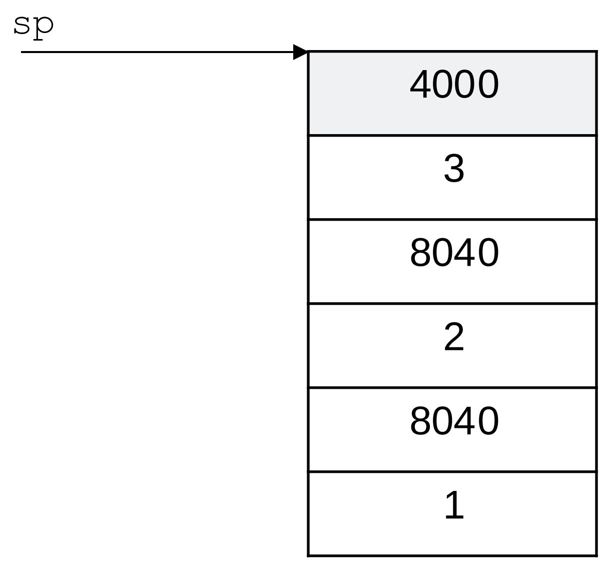
Instructions 14 - 15

```
mul a0, t0, a0               # return n * fact (n - 1)
jalr zero, ra, 0
```

- **Return to __fact()______ with a0 = __2_____**
- **New PC = _8040______**

Instructions 11 – 13

```
lw t0, 0(sp)      # return from jal: restore argument n
lw ra, 4(sp)      # restore the return address
addi sp, sp, 8    # adjust stack pointer to pop 2 items
```



- **a0 = 2 (from returned call)**
- **t0 = 3**
- **ra = 4000**
- **What is left in the stack after the execution of the instructions above?**

58

Instructions 14 - 15

```
mul a0, t0, a0    # return n * fact (n - 1)
jalr zero, ra, 0 # return to caller
```

- **Return to ___main()___ with a0 = ___6____**
- **New PC = _4000_**

Recursive Procedure Call Example: Fibonacci Numbers

- **The Fibonacci numbers are defined as follows:**

  F(n) = F(n – 1) + F(n – 2),
  F(0) and F(1) are defined to be 1

- **Rewriting this in C :**

```
int fib(int n) {
 if(n == 0) { return 1; }
 if(n == 1) { return 1; }
 return (fib(n - 1) + fib(n - 2));
}
```

Approaching the RISC-V Code

- **Decide what goes on the stack, write the prologue**
  - Allocating stack
  - Saving initial values
- **Write the epilogue**
  - Restore values from the stack
  - Release the stack
  - Return to the caller
- **Write the body– carefully**
  - This matches the C function body

Fibonacci Numbers (Prologue)

- **Space for three words on the stack**
  - return addr (ra)
  - n (a0)
  - temp sum (s0)
- **RISC-V code:**

```
fib:
   addi sp, sp, -12 # space for 3 words
   sw ra, 8(sp)     # save return address
   sw s0, 4(sp)     # save s0, used within
```

Fibonacci Numbers (Epilogue)

```
fib_done:
   lw   s0, 4(sp)     # restore s0
   lw   ra, 8(sp)     # restore ra
   addi sp, sp, 12    # pop the stack frame
   jalr zero, ra, 0   # return to caller
```

Fibonacci Numbers: Body Part 1

- **Start with the base cases**

```
if (n == 0) return 1;
if (n == 1) return 1;
```

- **RISC-V code:**

```
addi t1, a0, 0        # save a0 in t1
addi a0, x0, 1        # a0 = 1 (setup return value)
beq  t1, x0, fib_done # if n==0 goto fib_done
addi t0, x0, 1
beq  t1, t0, fib_done # if n==1 goto fib_done
```

Fibonacci Numbers (Body Part 2)

- **Next, the recursive case: careful, this is tricky**

```
return fib(n-1) + fib(n-2)
```

- **RISC-V code:**

```
addi a0, t1, -1     # a0 = n-1
sw   a0, 0(sp)      # need a0 after jal
jal  ra, fib        # fib(n-1)
add  s0, a0, 0      # save result
lw   a0, 0(sp)      # restore a0
addi a0, a0, -1     # a0 = n-2 (already n-1)
jal  ra, fib        # fib(n-2)
add  a0, a0, s0     # a0=fib(n-2)+fib(n-1)
                    # continues at fib_done...
```

fib.s

- **Breakpoint on jal's and jalr**
- **Watch a0, s0, sp, ra**
- **Single step the first jal to see ra change (on first call)**
  - Updated, but stays unchanged, on the recursive call
- **Single step the** jalr x0, ra, 0 **to see PC change (on last return)**
  - Updated, but stays unchanged, on the recursive returns

RISC-V Register Conventions

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

After-Class Activities

- **Draw out the stack showing the execution of fib.s with an input of 4 (the first 4 Fibonacci numbers will be printed)**

- **Compare with what RARS stack shows**