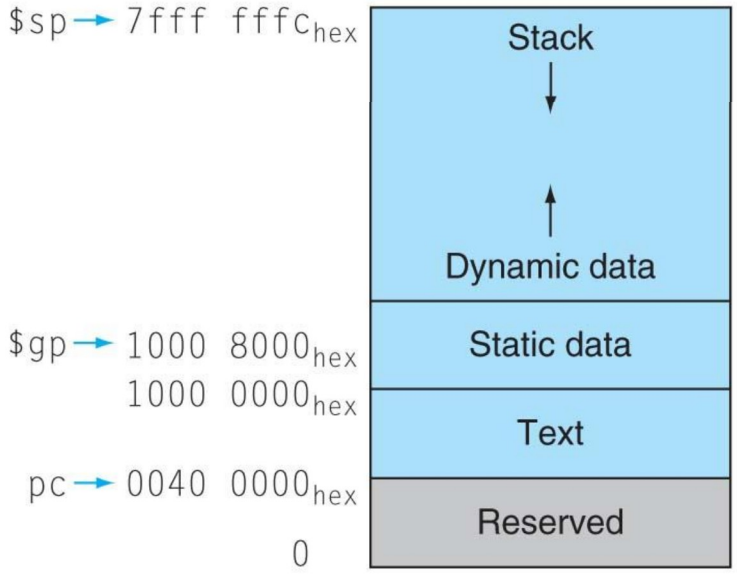
Nested Procedures

```
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

- CalleR: the calling function
- CalleE: 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 9th 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