ECE243

Prof. Enright Jerger

The NIOS II ISA

- **Memory**:
  - 32-bit address space
    - an address is 32bits
  - Byte-addressable
    - each address represents one byte
  - Hence: 2^{32} addresses = 2^{32}bytes = 4GB
  - Note: means NIOS capable of addressing 4GB
    - doesn't mean DE2 has that much memory in it
- **INSTRUCTION DATATYPES**
  - defined by the ISA (in C: unsigned, char, long etc)
  - byte (b) = 8 bits
  - Half-word (h) = 2 bytes = 16bits
  - word (w) = 4 bytes = 32 bits

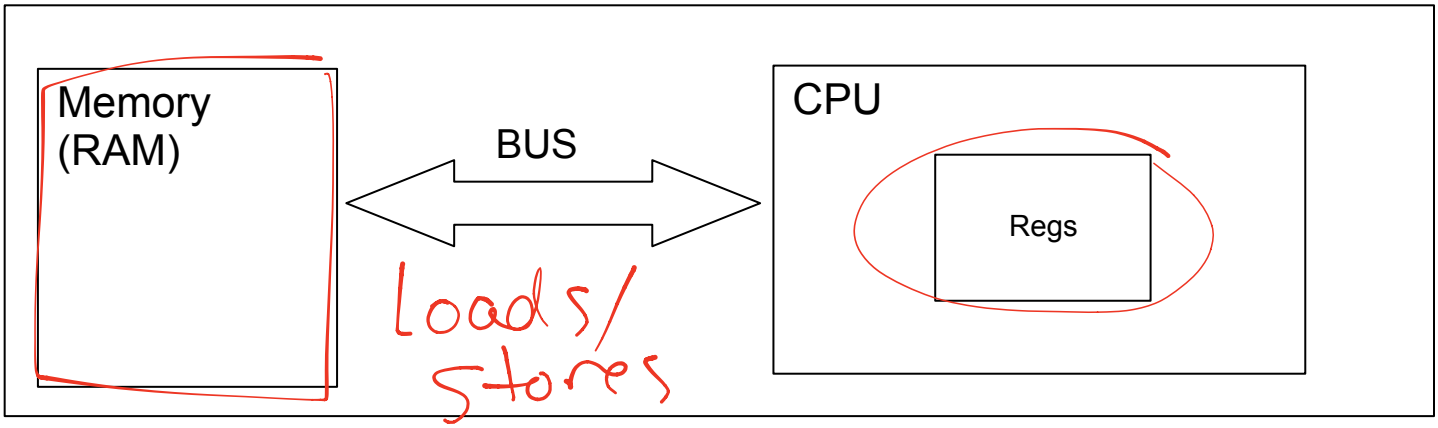
ALIGNMENT

- Processor expects:

- half-words and words to be properly aligned
- Half-word: address evenly divisible by 2
- Word: address evenly divisible by 4
- Byte: address can be anything
- Why? Makes processor internals more simple

- Ex: Load word at 0x27 ? No – will load @ 0x24
  Load halfword at 0x32 ? Yes
  Load word at 0x6 ? No – will load @ 0x4
  Load word at 0x14 ? Yes
  Load byte at 0x5 - Yes

REGISTERS



- An array of flipflops
  - managed as a unit
- Holds bits:
  - Can be interpreted as data, address, instr
- Are internal to the CPU
  - much faster than memory

- The PC is a register too
  - address of an instruction in memory

NIOS Registers

- 32bits each
- 32 general purpose registers: called r0-r31
- 6 control registers (learn more later)
- 1 PC: called pc

- GENERAL PURPOSE REGISTERS
  - r0: hardwired to always be zero 0x00000000
  - r8-r23: for your use
  - r1-r7, r24-r31: reserved for specific uses

more later

Common Operations

- **Math**:
  - add r8,r9,r10 # r8 = r9 + r10
  - sub r8,r9,r10 # r8 = r9 - r10
  - Also: mul, div
- **Logical**:
  - or r8,r9,r10 # r8 = r9 | r10
    - Example: 1010 | 0110 = 1110 bitwise OR

- Also: and, nor, xor

- **Copying**:
  - mov r8,r9 # r8 = r9 copy - doesn't erase r9

EXAMPLE PROGRAM 1

- C-code:
  - Unsigned char a = 0x0; # unsigned char==1byte
  - Unsigned char b = 0x1;
  - Unsigned char c = 0x2;
  - a = b + c;
  - c = b
- **Assume already init: r8=a, r9=b, r10=c:**

```
add r8, r9, r10 # r8=r9+r10 = 0x3
mov r10, r9 # r10 = r9 = 0x1
```

How to Initialize a Register

- movi instruction:
  - "move immediate"
  - movi r8,Imm16 # r8 = Imm16
- Imm16:
  - a 16-bit constant
  - called an "immediate operand"

- can be decimal, hex, binary
- positive or negative

EXAMPLE PROGRAM 1

- C-code:
  - Unsigned char a = 0x0; # unsigned char==1byte
  - Unsigned char b = 0x1;
  - Unsigned char c = 0x2;
  - a = b + c;
  - c = b
- **With Initialization:**

```
movi r8, 0x0    # r8 = 0x0
movi r9, 0x1    # r9 = 0x1
movi r10, 0x2   # r10 = 0x2

add r8, r9, r10
mov r10, r9
```

EXAMPLE PROGRAM 2

- Assume unsigned int == 4 bytes == 1 word
  - unsigned int a = 0x00000000;
  - unsigned int b = 0x11223344;
  - unsigned int c = 0x55667788;
  - a = b + c;

- With initialization:

program has 32-b.t values
can't use movi
movi can only handle 16-b.t imm

movia Instruction

- "move immediate address"
  - movia r9, Imm32 # r9 = Imm32
- Imm32
  - a 32-bit unsigned value (or label, more later)
  - doesn't actually have to be an address!

EXAMPLE PROGRAM 2

- Assume unsigned int == 4 bytes == 1 word
  - unsigned int a = 0x00000000;
  - unsigned int b = 0x11223344;
  - unsigned int c = 0x55667788;
  - a = b + c;
- Solution:

```
movia r8, 0x00000000
movia r9, 0x11223344
movia r10, 0x55667788
add r8, r9, r10 # r8 = r9 + r10
                = 0x11223344 + 0x55667788
```

MEMORY INSTRUCTIONS

- Used to copy to/from memory/registers
- Load: ldw rX, Imm16(rY)
  - Performs: Rx = mem[rY + Imm16]
- rY: holds the address to be accessed in memory
- Imm16:
  - sometimes called a 'displacement'
  - can be positive or negative
    - is actually a 2's complement number
  - Is added to the address in rY (but doesn't change rY)
- Store: stw rX, Imm16(rY)
  - performs: mem[rY + Imm16] = Rx

Types of Loads and Stores

- Load granularities:
  - ldw: load word; ldh: load halfword; ldb: load byte
- Store granularities:
  - stw: store word; sth: store halfword; stb: store byte
- load and store instructions that end in 'io'
  - eg: ldwio, stwio

- this means to bypass the cache (later) if one exists
- important for memory-mapped addresses (later)
  - otherwise same as ldw and stw

Example Program 3

- unsigned int a = 0x00000000;
- unsigned int b = 0x11223344;
- unsigned int c = 0x55667788;
- a = b + c;

- Challenge:
  - keep a,b,c in memory instead of registers

- Assume memory is already initialized:

Addr:	Value
0x200000:	0x00000000
0x200004:	0x11223344
0x200008:	0x55667788

Example Program 3

- unsigned int a = 0x00000000;
- unsigned int b = 0x11223344;
- unsigned int c = 0x55667788;
- a = b + c;

Memory:

Addr:	Value
0x200000:	0x00000000
0x200004:	0x11223344
0x200008:	0x55667788

- Solution:

movia r11, 0x200004
ldw r9, 0(r11) # r9 = mem[0+r11]
= mem[0x200004]
= 0x11223344

movia r11, 0x200008
ldw r10, 0(r11) # r10 = mem[0+r11] = mem[0x200008]
= 0x55667788

add r8, r9, r10 # r8 = r9 + r10 = 0x6688aacc

movia r11, 0x200000
stw r8, 0(r11) # mem[0+r11] = r8
mem[0x200000] = 0x6688aacc

* not using displacement - can we do better?

Optimized Program 3

- unsigned int a = 0x00000000;
- unsigned int b = 0x11223344;
- unsigned int c = 0x55667788;
- a = b + c;

Memory:

Addr:	Value
0x200000:	0x00000000
0x200004:	0x11223344
0x200008:	0x55667788

- Solution:

movia r11, 0x200000
ldw r9, 4(r11) # r9 = mem[4+r11] = mem[0x200004]
= 0x11223344

ldw r10, 8(r11) # r10 = mem[8+r11] = mem[0x200008]
= 0x55667788

add r8, r9, r10

stw r8, 0(r11) # mem[0+r11]=r8

mem[0x200000]=r8

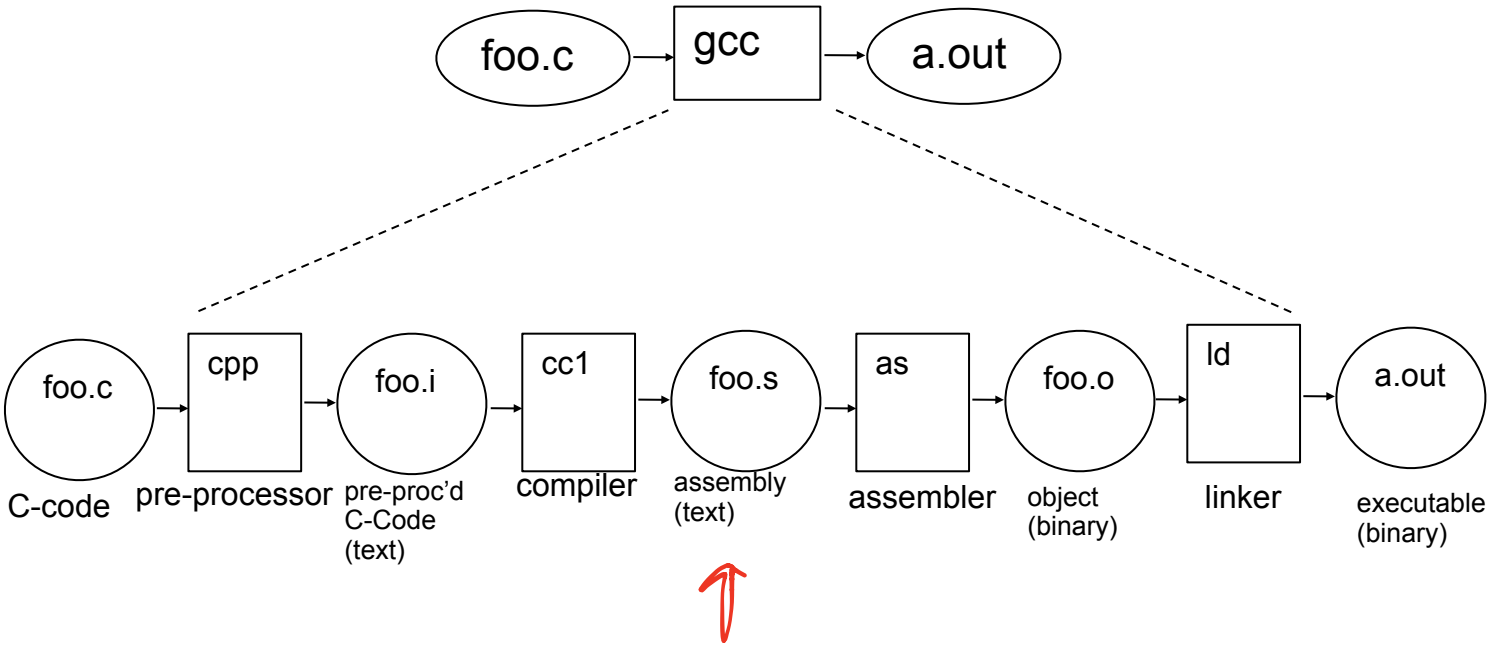
Addressing Modes

- How you can describe operands in instrs
- Addressing Modes in NIOS:
  - register: add r8, r9, r10
  - immediate: addi r8, r9, 0x25
  - register indirect with displacement: ldw r8, 0x4(r11)
  - register indirect: ldw r8, 0(r11)
- Note:
  - other more complex ISAs can have many addressing modes (more later)

ECE243

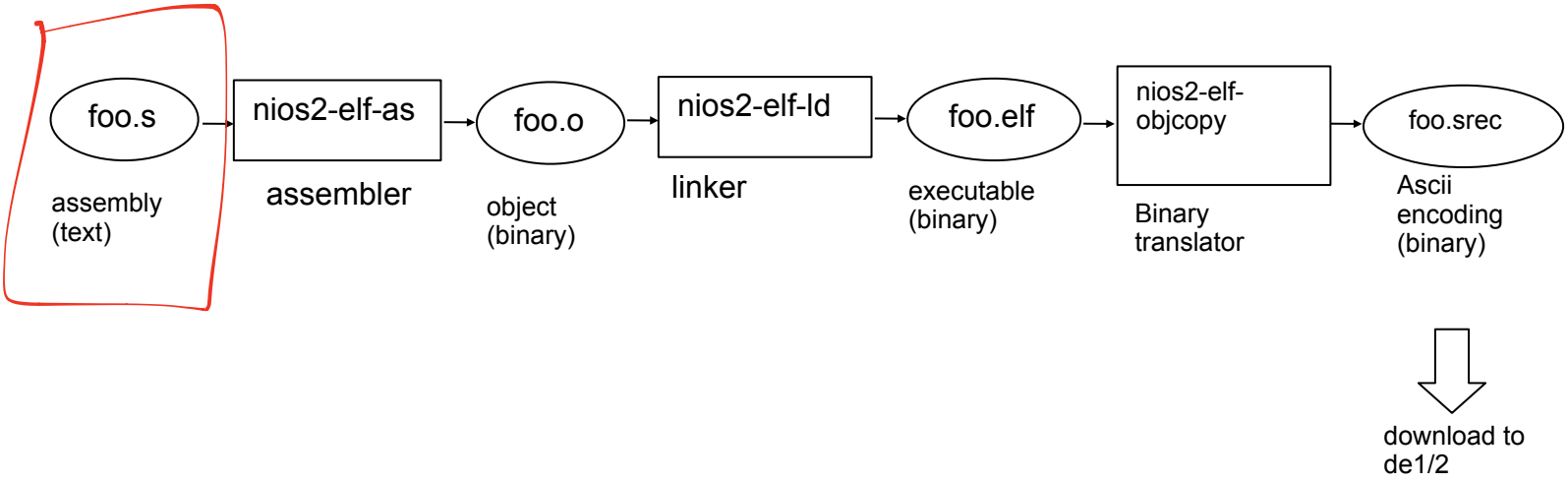
Assembly Basics

Typical GNU/Unix Compilation



- assembly:
  - text (human readable)
- linker:
  - can join multiple object files (and link-in libraries)
  - places instrs and data in specific memory locations

Compilation in ECE243



- in 243 you will mainly write assembly

WHAT IS *NOT* IN AN ASSEMBLY LANGUAGE?

- classes, hidden variables, private vs public
- datatype checking
- data structures:
  - arrays, structs
- control structures:
  - loops, if-then-else, case stmts
- YOU'RE ON YOUR OWN!

Assembly File: 2 Main parts:

- text section
  - declares procedures and insts
- data section
  - reserves space for data
  - can also initialize data locations
  - data locations can have labels that you can refer to

Example Assembly file:

```
.section .data
...
.section .text
...
```

Ex: Assembly for this C-Code:

```
unsigned int a = 0x00000000;
unsigned int b = 0x11223344;
unsigned int c = 0x55667788;
a = b + c;
```

Ex: Data Section

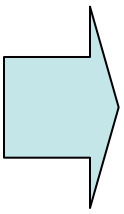
```
.section .data
.align 2   # means following must be aligned @ 2^2 = 4 byte boundary
va:  .word 0
vb:  .word 0x11223344
vc:  .word 0x55667788
```

label

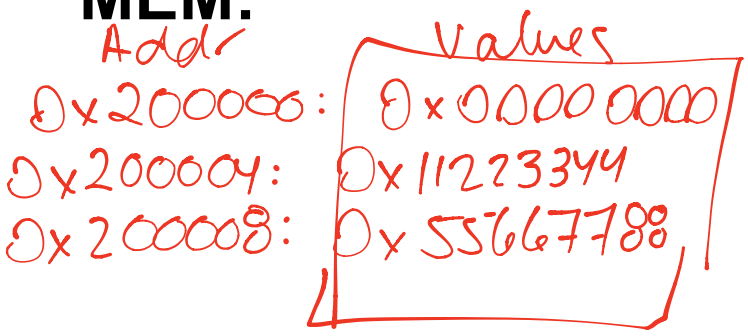
size - .word, .hword, .byte

Ex: When Loaded into Memory

```
        .section .data
        .align 2
va:     .word 0
vb:     .word 0x11223344
vc:     .word 0x55667788
```



MEM:



- linker decides where in mem to place data section
  - lets assume it places it starting at 0x200000
- .section and .align are 'directives'
  - they tell assembler to do something
  - but they don't become actual instructions!
- the labels and .word are also gone
  - only the data values are now in memory

Ex: Assembly for this C-Code:

```
unsigned int a = 0x00000000;
unsigned int b = 0x11223344;
unsigned int c = 0x55667788;
a = b + c;
```

Ex: Text Section

USEFUL ASSEMBLER DIRECTIVES AND MACROS

- /* comments in here */
- # this comments-out the rest of the line
- .equ LABEL, value
  - replace LABEL with value wherever it appears
  - remember: this does not become an instruction
- .asci “mystring” # declares ascii characters
- .asciz “mystring” # ends with NULL (0x0)

Arrays and Space

- Myarray: .word 0, 1, 2, 3, 4, 5, 6, 7
  - declares an array of 8 words
  - starts at label ‘Myarray’
  - initializes to 0,1,2,3,4,5,6,7
- myspace: .skip SIZE
  - reserves SIZE bytes of space

  - starts at label ‘myspace’
  - does not initialize (eg., does not set to zero)
- myspace: .space SIZE
  - same as .skip but initializes all locations to 0

Ex: Arrays and Space

- Create an array of 4 bytes at the label myarray0 initialized to 0xF,0xA,0xC,0xE

- Reserve space for an 8-element array of halfwords at the label myarray1, uninitialized

- Reserve space for a 6-element array of words at the label myarray2, initialized to zero

Understanding Disassembly

- can run “make SRCS=foo.s disasm”
- will dump a “disassembly” file of program

- shows bare real instructions
  - all assembler directives are gone
- **Example disasm output:**

```
01000000 <main>:
01000000:        02c04034  movi     r11,256
01000004:        5ac01804  addi     r11,r11,96
```

ECE243

Control Flow

HOW TO IMPLEMENT THIS?:

```
for (i=0;i<10;i++){
    …
}
```

- **NEED:**
  - a way to test when a condition is met
  - a way to modify the PC
    - ie., not just execute the next instruction in order
    - ie., do something other than PC = PC + 4

UNCONDITIONAL BRANCHES

- Can change the PC
- Starts execution at specified address

- *Unconditional* branch: br

br LABEL

- does: PC = LABEL
- *unconditional:* always!

Example:

- Jump: jmp

jmp rA

- does: PC = rA
- is also unconditional

Example:

Conditional branches

- only branch if a certain condition is true

bCC rA, rB, LABEL # if rA CC rB goto LABEL

- does signed comparisons, ie. you can use negative numbers

- CC:
  - eq (=), gt (>), ge (>=), lt (<), le (<=), ne (!=)

Example:

Ex: make these loops

(assume r8 initialized)

decrement r8 by 1, loop-back if r8 is non-zero,

```
LOOP: subi r8, r8, 1   # r8=r8-1
      bne  r8, r0, LOOP
```

increment r8 by 1, loop-back if r8 is equal to r9

```
LOOP: addi r8, r8, 1   # r8 = r8 + 1
      beq r8, r9, LOOP
      ...
```

increment r8 by 4, loop-back if r8 less than r9

```
LOOP: addi r8, r8, 4
      blt r8, r9, LOOP
```

decrement r8 by 2, loop-back if r8 greater-than-eq-to zero

```
LOOP: subi r8, r8, 2
      bge r8, r0, LOOP
```

Examples

```
If (r8 > r9) {
    …    // THEN
} else {
    …    // ELSE
}
 // AFTER
```

```
      bgt r8, r9, THEN
ELSE: ...
      br AFTER
THEN: ...
AFTER: ...
```

```
If (r8 == 5) {
    …    // THEN
```

```
      movi r11, 5
      bne r8, r11, ELSE
THEN: ...
```

```
} else {
    …    // ELSE
}
```

```
br AFTER
ELSE: ...
AFTER: ...
```

```
If (r8 > r9 && r8 == 5) {
    …    // THEN
} else {
    …    // ELSE
}
```

```
ble r8, r9, ELSE
movi r10, 5
bne r8, r10, ELSE
...
br AFTER
THEN: ...
ELSE: ...
AFTER: ...
```

```
If (r8 <= r9 || r8 != 5) {
    …    // THEN
} else {
    …    // ELSE
}
```

```
ble r8, r9, THEN
movi r10, 5
bne r8, r10, THEN
ELSE: ...
br AFTER
THEN: ...
AFTER: ...
```

```
for (r8=1; r8 < 5; r8++) {
        r9= r9 + r10;
}
```

```
        movi r8, 1
        movi r11, 5
Loop:   bge r8, r11, AFTER
Body:     add r9, r9, r10
          addi r8, r8, 1
           br Loop
After:    ...
```

```
while (r8 > 8){
        r9= r9 + r10;
        r8--;
}
```

```
        movi r11, 8
Loop:   ble r8, r11, AFTER
Body:   add r9, r9, r10
        subi r8, r8, 1
        br Loop
After:  ...
```

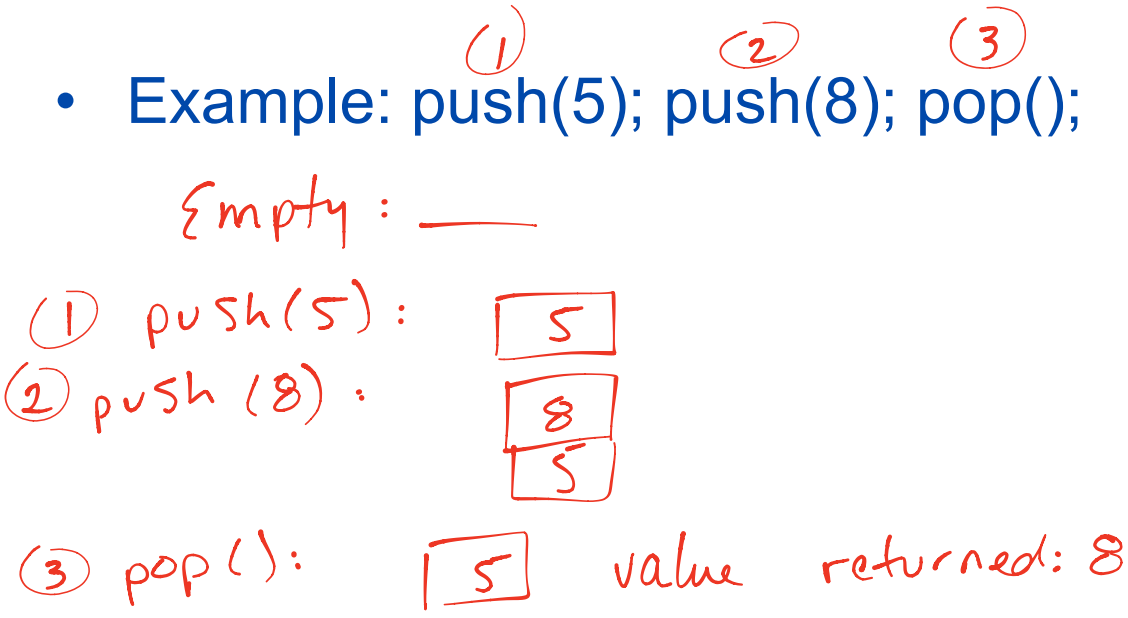
ECE243

Stacks

STACKS

- LIFO data structure (last in first out)
- Push: put new element on 'top'
- Pop: take element off the 'top'

- Example: push(5); push(8); pop();



A STACK IN MEMORY

- pick an address for the "bottom" of stack
  - stacks usually grow "upwards"
  - i.e., towards lower-numbered address locations
- programs usually have a "program stack"

- for use by user program to store things

- NIOS:
  - register r27 is usually the “stack pointer” aka ‘sp’
  - you can use ‘sp’ in your assembly programs
    - to refer to r27
  - the system initializes sp to be 0x17fff80

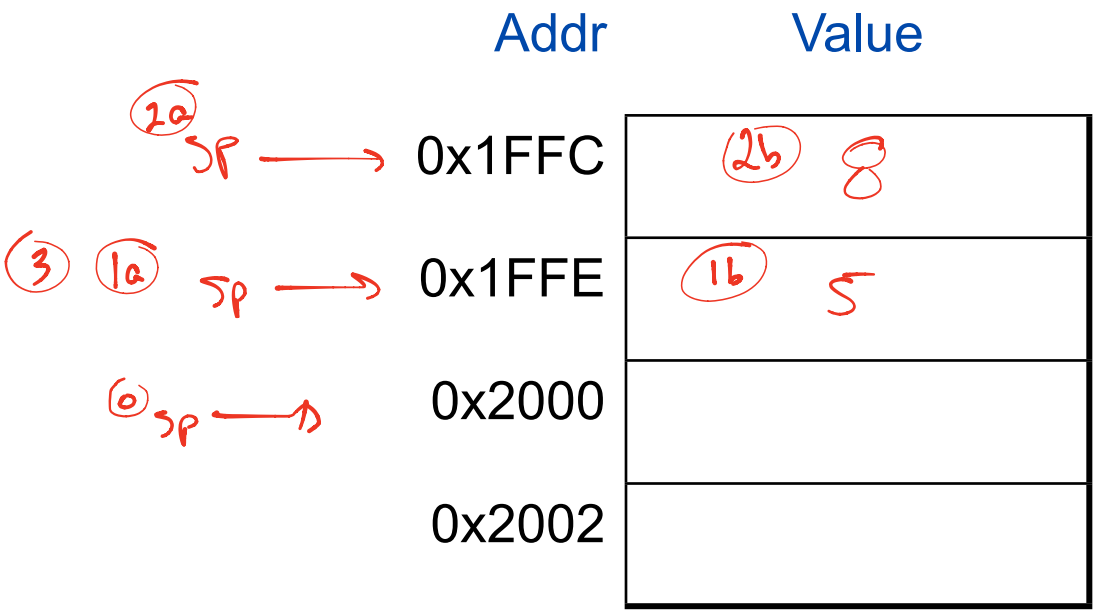
Ex: Stack in Memory of Halfwords

- Example: push(5); push(8); pop();

sp = 0x2000

movia sp, 0x2000

decrement sp, then store value



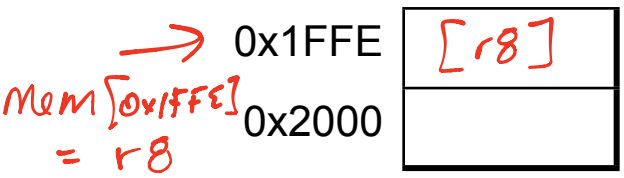
ASSEMBLY For Stack of Halfwords

- initialize stack pointer to 0x2000 (bottom of stack)

movia sp, 0x2000

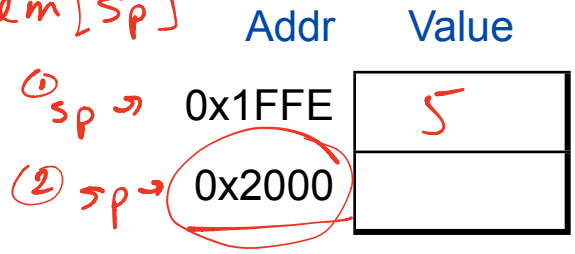
- Push: assume we want to push halfword in r8

```
subi sp, sp, 2
sth r8, 0(sp)   # mem[0x1FFE] = r8
```



- Pop: assume we want to pop halfword into r8

```
(1) ldh r8, 0(sp)   # r8 = mem[sp]
(2) addi sp, sp, 2
```



- Note: we grow then push, pop then shrink
  - by convention
  - could it be the other way?
- Note2: you don't actually delete the value
  - it is still there!

ECE243

Subroutines

SUBROUTINES

```
void bar(){
    return;
}

void foo(){
    bar();
}
```

- foo calls bar, bar returns to foo

SUBROUTINE CALLS vs BRANCHES

- a branch replaces the value of the PC
  - with the new location to start executing
  - so does a subroutine call
- a subroutine call "remembers where it came from"
  - how?

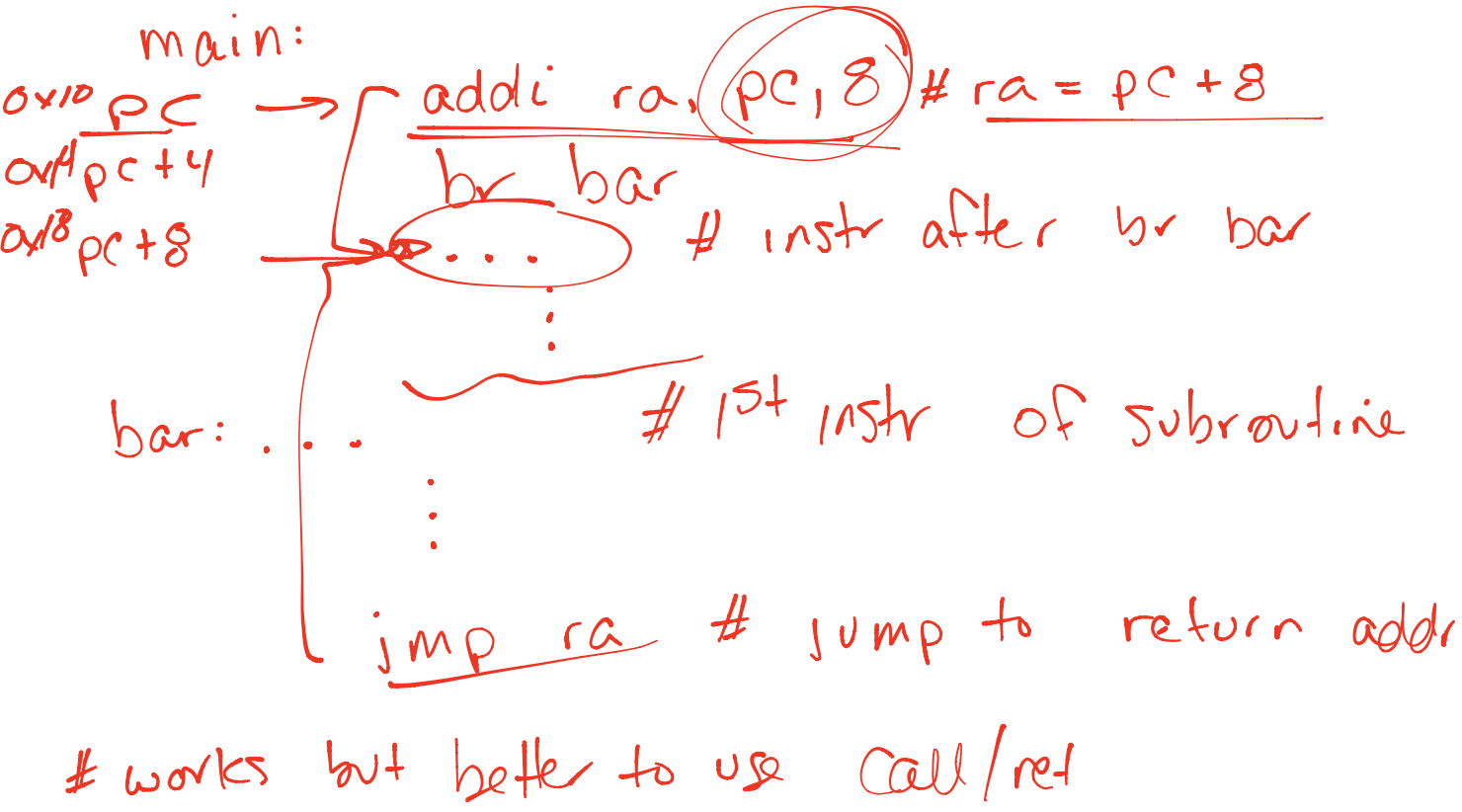
Store PC in register
· works for 1 call
push previous PC on stack in mem

RETURN ADDRESS REGISTER

- r31 is the return address register (aka ra)
  - by NIOS convention
  - you can use 'ra' in assembly
- NIOS convention for managing return addr:
  - push the previous return address (ra) on the stack
  - save the most recent return address in ra

Make A Subroutine Call "by hand"

Have main call bar and bar return to main



call and ret instructions

- call LABEL
  - does two steps in one instruction:

ra = pc+4 # ra points to instruction after the call

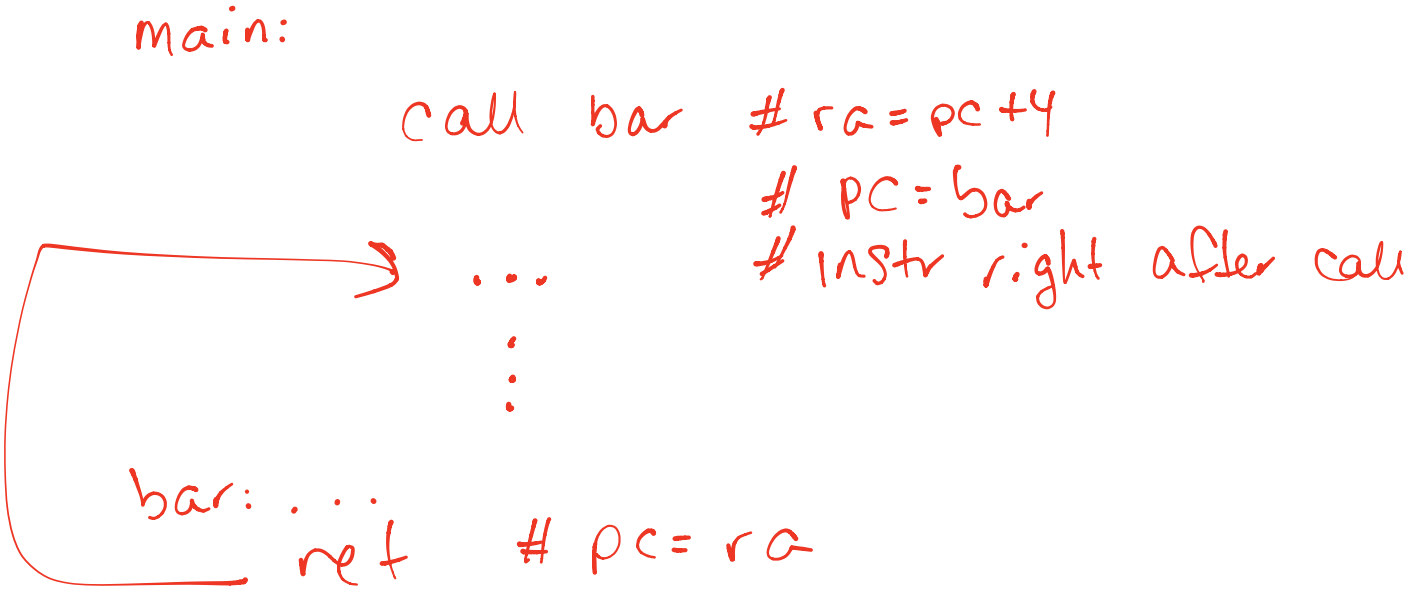
pc = LABEL # branch to the call target location

- ret
  - does the same thing as jmp ra:

    pc = ra

Subroutine Call with call/ret

Have main call bar and bar return to main



More SUBROUTINES

```
void car(){
    return;
}
void bar(){
    car();
}
```

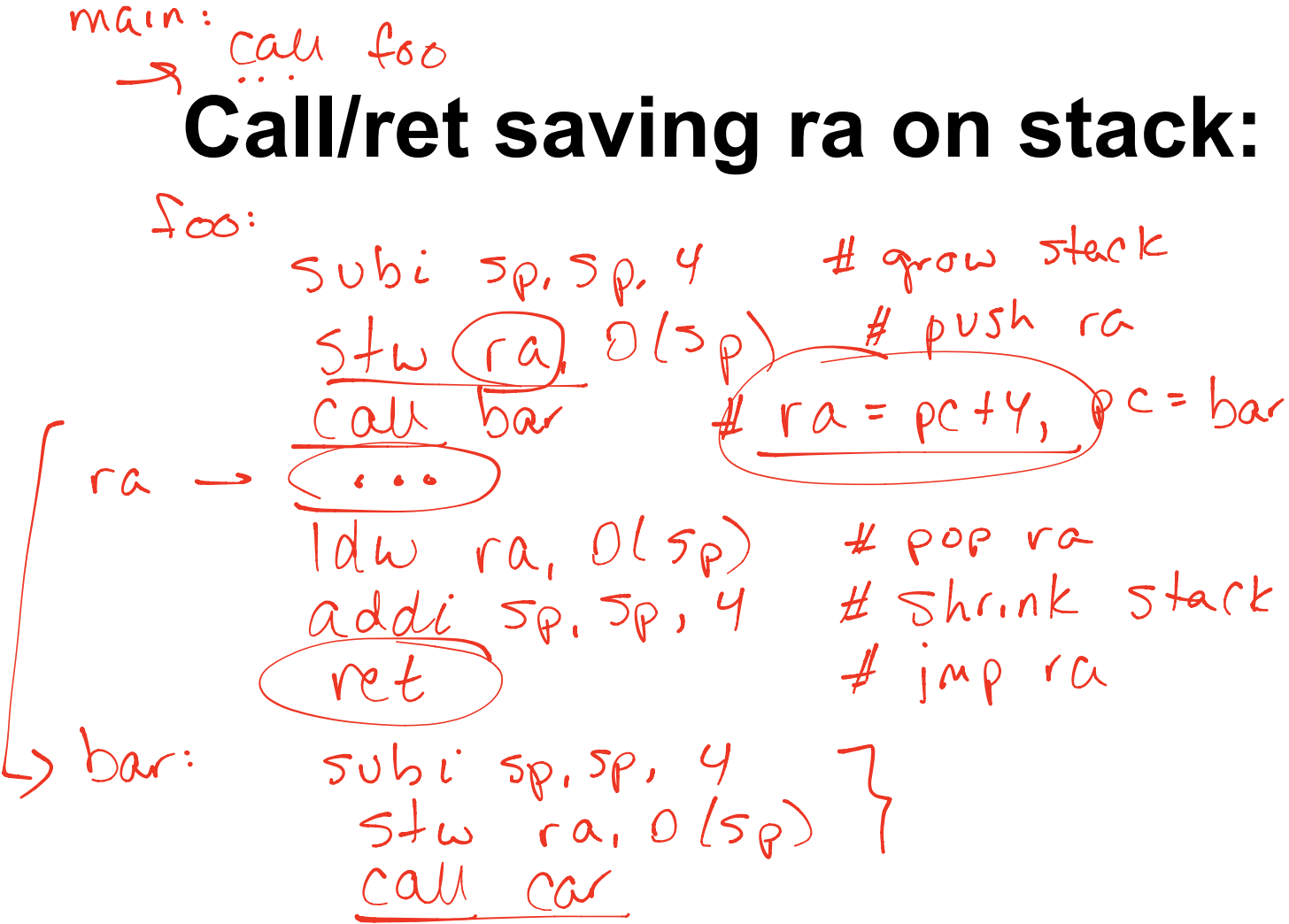
```
void foo(){
    bar();
}
```

- foo saves return address in ra, calls bar
- bar saves return address in….uh oh!

Handling Multiple Nested Calls

- Before you call anybody:
  - Push the return address on the stack
- When you are done calling:
  - Pop the return address off the stack

Call/ret saving ra on stack:

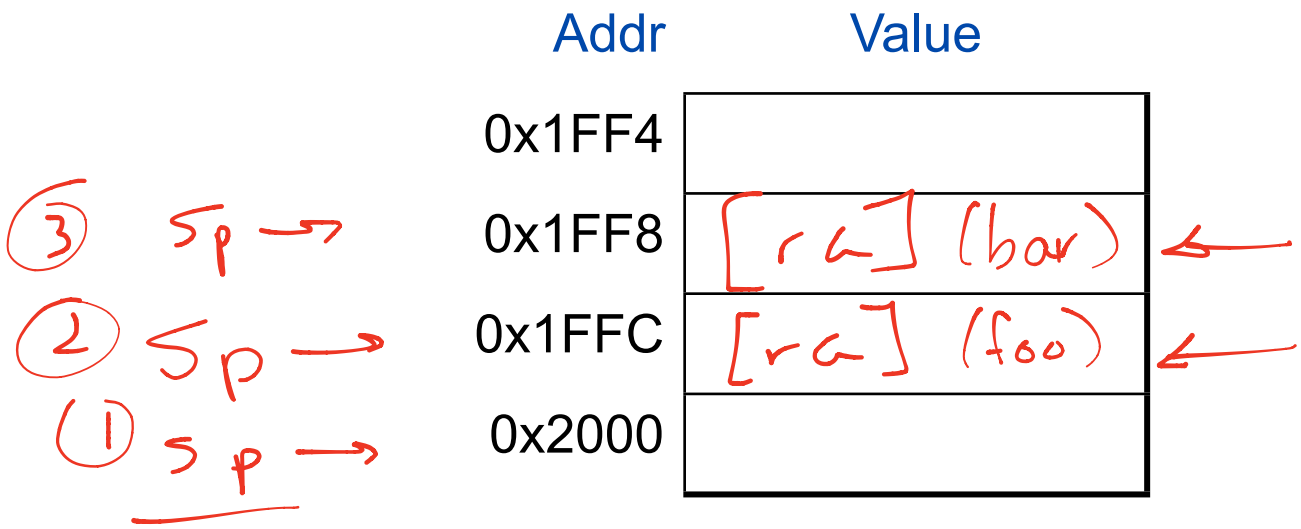


→ ldw ra, 0(sp)
addi sp, sp, 4
ret

car: ret

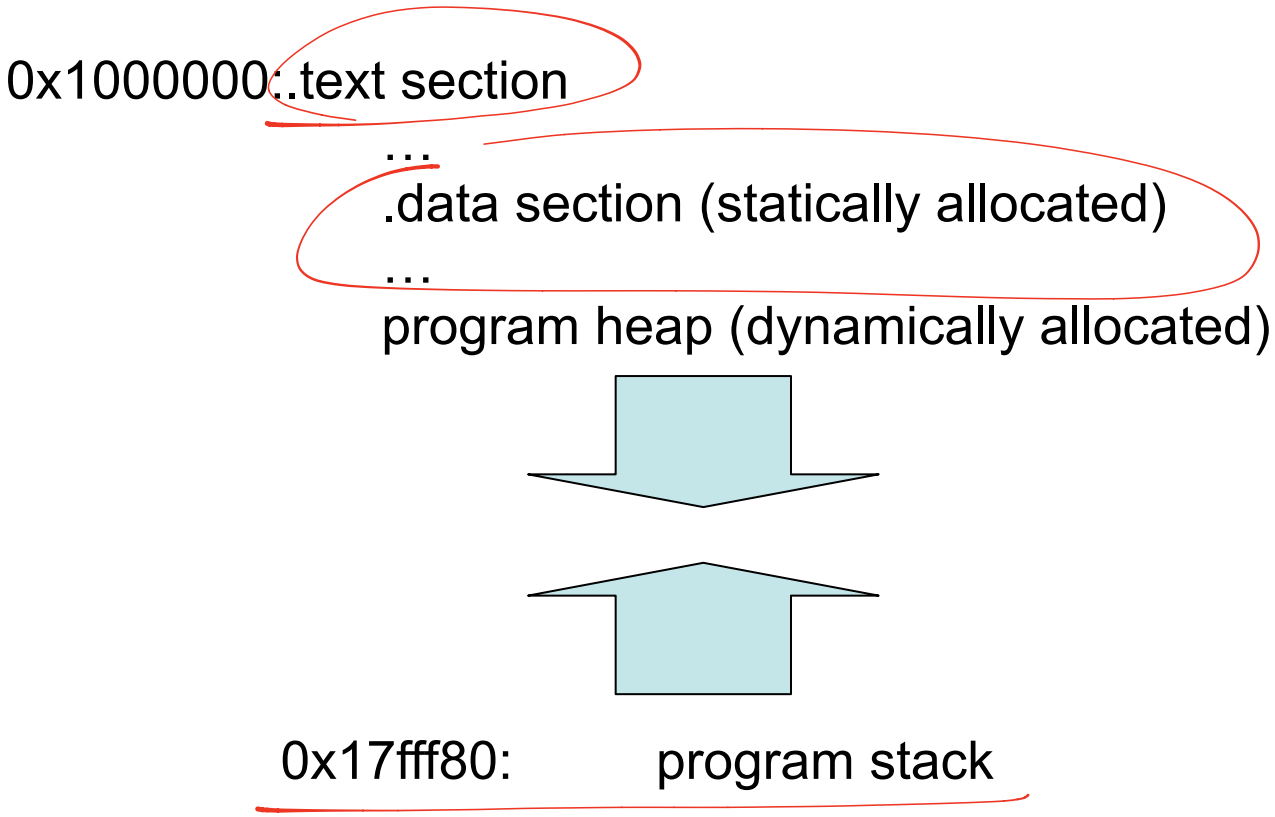
Visualizing Stack in Memory

- assume stack originally initialized to 0x2000
- foo pushes ra, calls bar; bar pushes ra, calls car



- NOTE:
  - to be correct, your "main" routine should push/restore ra
  - if you want to be able to return successfully from main

NIOS Memory Use:



CALLER AND CALLEE

```
void car(){
    return;
}
void bar(){
    car();
}
void foo(){
    bar();
}
```

← bar is the caller
car is callee

← foo is caller
bar is callee

INDEPENDENCE OF SUBROUTINES

- a big program may have many subroutines
- subroutines all need registers
  - there are only 32 registers (fewer free for use)
  - how do we arrange for subrs to all share regs?
- solution1: each subr can use certain regs
  - hard to manage, what if things change
  - will run out of registers with a big program
- solution2: subrs share the same regs
  - must save and restore register values
  - two schemes for deciding who saves/restores

SOLUTION2a: CALLER-SAVE

- the caller saves registers it cares about
  - needn't save a reg value you no longer need

```
main:
    subi sp,sp,4      # save ra
    stw ra,0(sp)
    movi r8, 0x32     # value in r8
    …                 # code that uses r8

    subi sp, sp, 4
    stw r8, 0(sp)
    call foo
    ldw r8, 0(sp)
    addi sp, sp, 4
```

Addr	Value
0x1FF4	
0x1FF8	0x32 [r8]
0x1FFC	[ra] (main)
0x2000	

```
…                   # code that uses r8
ldw ra,0(sp)        # restore ra
addi sp,sp,4
ret
```

Nios Convention

- registers r8-r15 are caller saved
- if you want r8-r15 to live across a call site
  - you must save/restore it before/after any call
  - because the callee might change it!
- **You should do this for all code you write!**
  - Even if only one callee
  - Even if you know it won't modify r8-r15
  - You might add more callees later
  - Your TA might deduct marks for bad style ☹

Caller Save: bigger example

```
main:
    …               # save ra
    …               # code using r8, r9, r10
    …
```

```
subi sp, sp, 12  # grow stack by 3x4B
stw r8, 0(sp)
stw r9, 4(sp)
```

```
    stw  r10, 8(sp)
    call foo
    ldw  r8, 0(sp)
    ldw  r9, 4(sp)
    ldw  r10, 8(sp)
    ... addi sp, sp, 12
    ...          # code still using r8,r9,r10
    ...          # restore ra
    ret
```

	Addr	Value
sp →	0x1FF0	[r8]
	0x1FF4	[r9]
	0x1FF8	[r10]
→	0x1FFC	[ra]
→	0x2000	

Solution2b: Callee Save

- Save/restore all callee-saved regs at beginning/ end of subroutine

```
foo:
    subi sp, sp, 4
    stw  r16, 0(sp)

    ...                  # some code

    movi r8, 0x393       # messes up r8      # r8 is caller saved

    movi r16, 0x555      # messes up r16 - callee saved
    ...                  # other code

    ldw  r16, 0(sp)      # pop r16
    addi sp, sp, 4       # shrink stack
```

	Addr	Value
	0x1FF4	
	0x1FF8	
	0x1FFC	[r16]
→	0x2000	

ret

Nios Convention

- registers r16-r23 are callee saved
- if you want to modify r16-r23
  - then you must save/restore them at the beginning/end of your procedure
- **You should do this for all code you write!**
  - Even if only one caller
  - Even if you know it doesn't care about r16-r23
  - You might add more callers later
  - Your TA might deduct marks for bad style ☹

Summary

- ~~r8-r15~~: callee-saved
  - save these before you use them!
  - restore them when you are done
  - recommend: save them at the top, restore at bottom
- ~~r16-r23~~: caller-saved
  - save these before you make a call
  - restore them right after the call
- Both:
  - only have to save/restore the regs that you modify

Strategy for Managing Registers

- for temporaries or subr's that don't call anybody
  - use only r8-15 (caller-save)
  - don't have to save/restore them in this case!
- otherwise:
  - use r16-r23 (callee-save)
  - save/restore them at the top/bottom

Returning a Value

```
int foo(){
    return 25;
}
```

- **NIOS Convention:**
  - r2 is used for returning values
  - Note1: therefore r2 must be caller-saved
    - Since callee can modify it with a return value
  - Note2: r3 can be used to return a 2nd value
    - not often used---advanced

Example

```
Main(){
```

```
main:
    # save ra
```

```
   …
   r8 = foo();
   …
}

int foo(){
   return 25;
}
```

```
# save any r8-r15 that I care about
call foo
mov r8, r2   # ret val in r2
# restore r8-r15 that I saved
# restore ra
ret

foo:
   movi r2, 25
   ret   # pc = ra
```

Passing Parameters

- **POSSIBILITIES:**
  - put value(s) into registers (USED)
  - put value(s) into predetermined mem location(s)
    - like a global variable (NOT USED)
  - push/pop value(s) on/off stack (USED)
- **NIOS Convention:**
  - r4-r7 can be used to pass up to 4 parameters
  - use the stack if more than 4 parameters

Example

params

```
Int add2(int a, int b){
   Return a + b;
}

int main(){
   return add2(25,37);
}
```

```
add2:
        add r2, r4, r5
        ret

main:
        # save ra
        movi r4, 25
        movi r5, 37
        call add2
        # restore ra
        ret
```

REGISTER USAGE SUMMARY:

- r0: hardwired to zero
- r2,r3: return value registers (caller save)
- r4-r7: subroutine parameters (caller save)
- r8-r15: general use (caller-save)
- r16-r23: general use (callee-save)
- r27: sp
- r31: ra

- more later on r1,r24-26,r28-30

Is ra Caller or Callee Saved?

ra is Caller Saved

```
foo:
  …
  …
  …
  ret
```

```
foo:
  # save ra
  …
  call bar
  …
  # restore ra
  ret
```

- caller should save a caller-saved reg that it cares about across any call site
- eg: foo cares about ra, should save it if it is making any calls (i.e., to bar)
  - it seems like ra is caller-saved

ra is Callee Saved

```
foo:
  …
  …
  …
  ret
```

```
foo:
  # save ra
  …
  call bar      # ra = pc+4
                # pc = bar
  …
  # restore ra
  ret
```

- callee should save any callee-saved reg that it plans to modify
- eg: the call instruction will modify ra (inside foo), hence foo should save/restore it
  - therefore ra is callee-saved

Conclusion

- There are arguments for both:
  - ra is caller-saved
  - ra is callee-saved
- callee-saved has the stronger argument
  - ra is treated most like a callee-saved register

Bigger Example

```
Int add6(int a, int b, int c, int d, int e, int f){
   Return add2(a,b) + add2(c,d) + add2(e,f);
}

int main(){
   return add6(11,22,33,44,55,66);
}
```

```
int main(){
  return add6(11,22,33, 44,55,66);
}
```

r4 - r7 stack

```
main:
    subi sp, sp, 8
    stw ra, 4(sp)
    stw r16, 0(sp)

    movi r4, 11
    movi r5, 22
    movi r6, 33
    movi r7, 44
    subi sp, sp, 8   # grow for params e,f
    movi r16, 55
    stw r16, 0(sp)   # params right to left on stack, right = bottom
    movi r16, 66
    stw r16, 4(sp)
    call add6
    addi sp, sp, 8   # shrink de-alloc space alloc'd pre-call

    ldw ra, 4(sp)
    ldw r16, 0(sp)
    addi sp, sp, 8
    ret
```

Addr	Value
0x1FF0	55
0x1FF4	66
0x1FF8	[r16]
0x1FFC	[ra] (main)
0x2000	

-8

```
Int add6(int a, int b, int c, int d, int e, int f){
  return add2(a,b) + add2(c,d) + add2(e,f);
}
```

```
add6:
    subi sp, sp, 8
    stw ra, 4(sp)
```

```
stw  r16, 0(sp)
call add2  # r4, r5 - 1st 2 params
mov  r16, r2
mov  r4, r6
mov  r5, r7   # next 2 params
call add2  # adds c+d
add  r16, r16, r2
ldw  r4, 8(sp)
ldw  r5, 12(sp)
call add2  # e+f
add  r2, r16, r2
ldw  r16, 0(sp)
ldw  ra, 4(sp)
addi sp, sp, 8
ret
```

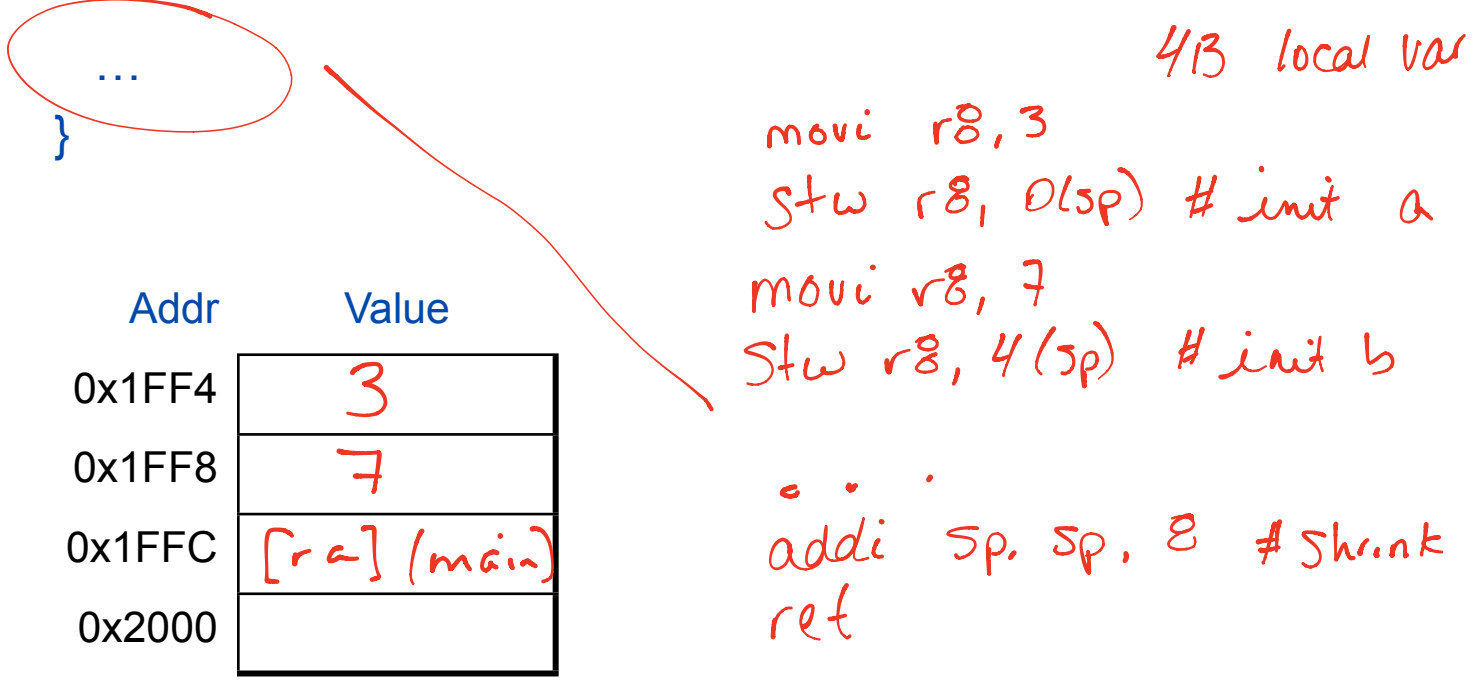
Addr	Value
0x1FE0	
0x1FE4	
0x1FE8	[r16]
0x1FEC	[ra] (add16)
0x1FF0	55
0x1FF4	66
0x1FF8	[-16]
0x1FFC	[ra] (main)
0x2000	

Local Variables

```
main(){
  …
  foo(5);
  …
}
void foo(int x){
  Int a = 3;
  Int b = 7;
```

```
main:
        # save ra
        movi r4, 5
        call foo
        # restore ra
        ret

foo:
        subi sp, sp, 8  # space for 2x
```



Subroutine Convention Summary

```
foo:
    #PROLOGUE
    #(1) grow stack to make space for (2) – (4)
    #(2) save ra (if making any calls)
    #(3) save callee-save registers (if planning to use any)
    #(4) initialize local variables (if any)

    #PRE-CALL
    #save caller-save registers with in-use values (if any)
    #push parameters (if more than four)

    call bar

    #POST-CALL
    #pop parameters (if > four)
    #restore caller-save registers (if any)

    #EPILOGUE
    # restore callee-save registers used (if any)
```

```
# restore ra (if calls made)
# shrink stack (by amount allocated in (1))
ret
```