From C to MIPS

David E. Culler

CS61CL

Sept 16, 2009

Lecture 4

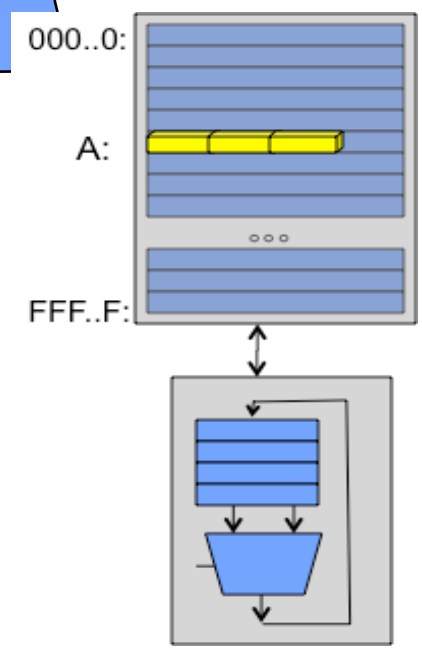
Review

- Arrays, Structs, and Pointers allow you define sophisticated data structures
  - Compiler protects you by enforcing type system
  - Avoid dropping beneath the abstraction and munging the bits
- All map into untyped storage, ints, and addresses
- Executing program has a specific structure
  - Code, Static Data, Stack, and Heap
  - Mapped into address space
  - "Holes" allow stack and heap to grow
  - Compiler defines what the bits mean by enforcing type
    - » Chooses which operations to perform
- Poor coding practices, bugs, and architecture limitations lead to vulnerabilities

Today

```
int main() {
  …
}
```

```
.data
A:.word 5
…
.text
main: lw $a0, x
jal decr
move $a0,$v0
```



Elements of the Language

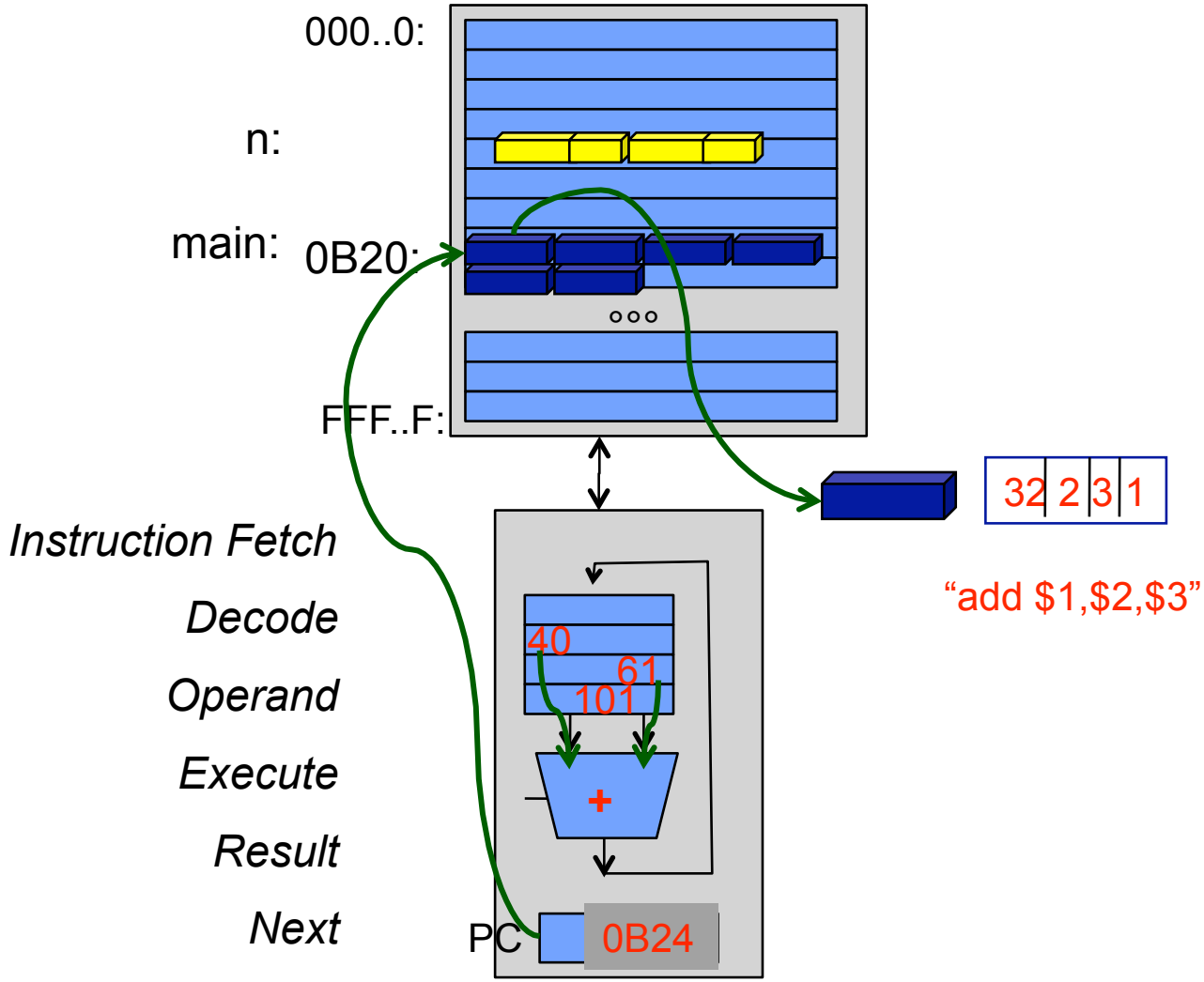
- Basic Data Types: char, int, float double
- Type Constructors: array, struct, pointer
- Variables
- Expressions
- Sequence of statements
- Conditionals
- Iteration
- Functions

What does the machine do?

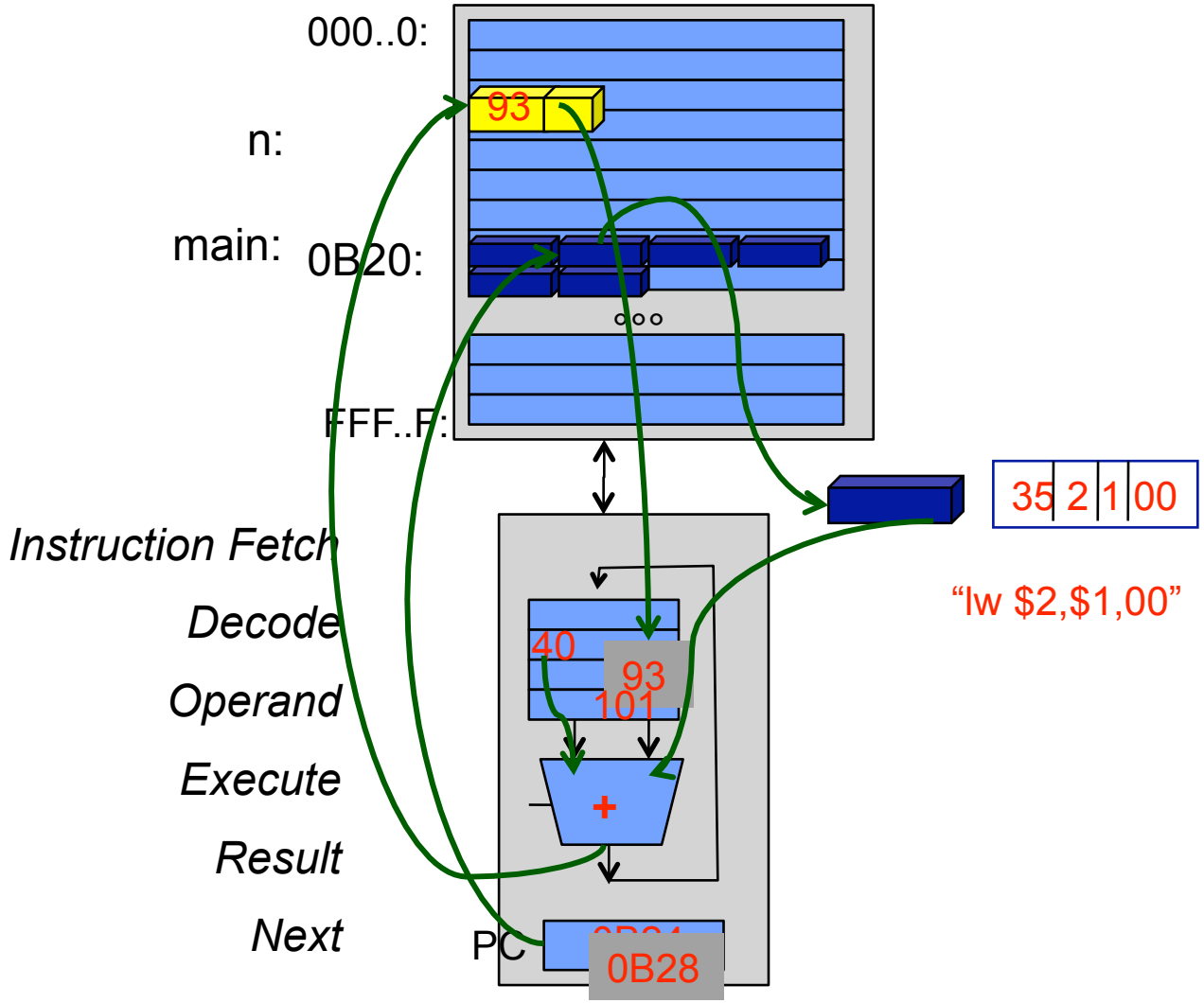
Instruction Execution Cycle

- Instruction Fetch
- Decode
- Operand Fetch
- Execute
- Result Store
- Next Instruction

Instruction Cycle



Instruction Cycle - again



What does the machine do?

Instruction Execution Cycle

- **Instruction Fetch**
- **Decode**
- **Operand Fetch**
- **Execute**
- **Result Store**
- **Next Instruction**

Register Transfers

inst <= mem[PC]

op, rd, rs, rt <= inst

A <= reg[rs]
B <= reg[rt]

R <= A + B

reg[rd] := A + B

PC := PC + 4

MIPS Assembly Language (MAL)

segments

values

```
        .data
A:      .word 5
        .word 6
        …
        .text
main:   la $t0, A
        lw $a0, 4($t0)
        jal decr
        move $a0,$v0
        …
decr:   addi $v0, $a0, -1
        jr $ra
```

label

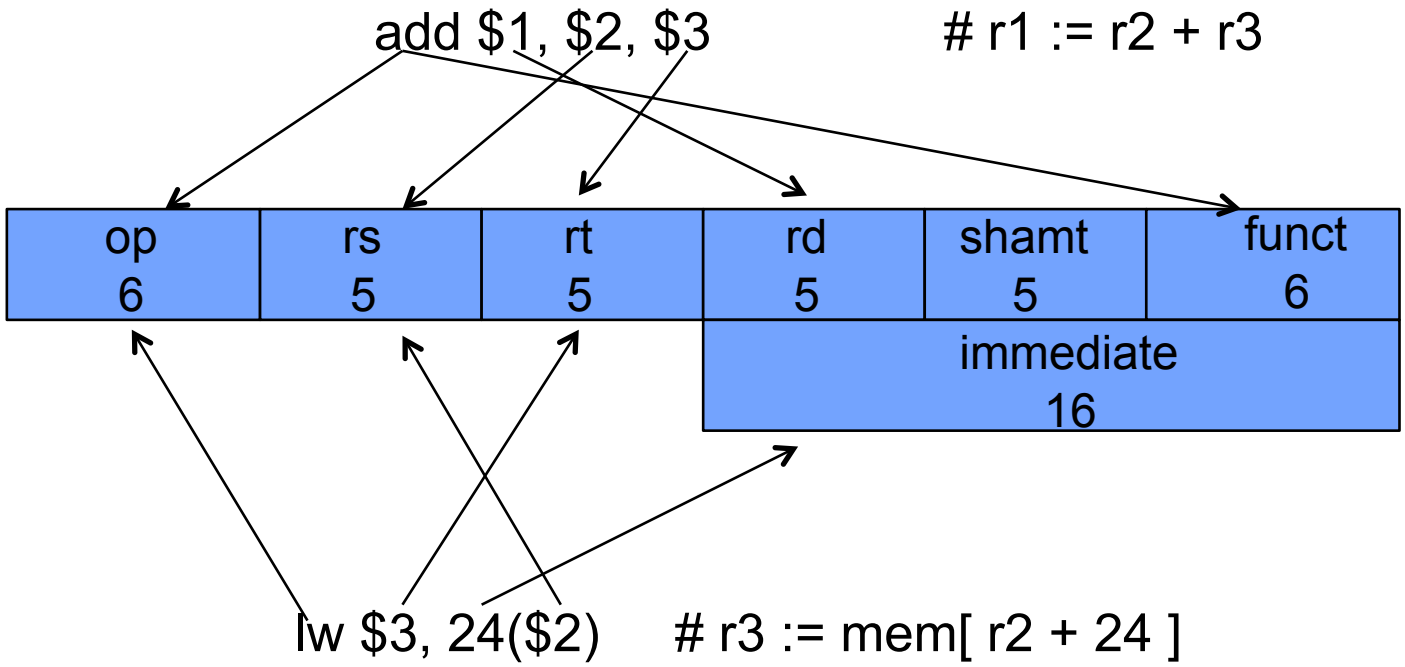
operands

opcode

registers

literals

MIPS Instruction Format



MIPS register conventions

Name	Number	Use	Callee must preserve?
$zero	$0	constant 0	N/A
$at	$1	assembler temporary	No
$v0–$v1	$2–$3	returns values	No
$a0–$a3	$4–$7	function arguments	No
$t0–$t7	$8–$15	temporaries	No
$s0–$s7	$16–$23	saved temporaries	Yes
$t8–$t9	$24–$25	temporaries	No
$k0–$k1	$26–$27	reserved for OS kernel	No
$gp	$28	global pointer	Yes
$sp	$29	stack pointer	Yes
$fp	$30	frame pointer	Yes
$ra	$31	return address	N/A

Administration

- Calendar with links on the home page
- Readers will be in lab 1 hour per week
- HW3R – Resubmit as hw3r permitted till 11:59 pm Saturday
  - as you learn, don't be afraid to make a fresh start
- HW4 out – all future HW will be W-W
  - less work than hw3, little reliance on Th/F Lab, start right away
  - Continues C concepts plus basic MAL
- Project 1 posted
- Mid Term 1 shifted to Wed 10/7 in class time
  - alternative Monday 10/5 @ 4 pm

Elements of the Language

- Basic Data Types: char, int, float double
- Type Constructors: array, struct, pointer
- Variables
- Expressions
- Sequence of statements
- Conditionals
- Iteration
- Functions

Expressions

```
y1 = (-b + sqrt(b*b - 4*a*c) / (2*a);
y2 = (-b - sqrt(b*b - 4*a*c) / (2*a);
```

```
t1 = -b;
t2 = b*b;
t3 = 4*a;
t4 = t3*c;
t5 = t2 – t4;
t6 = sqrt(t5);
t7 = t1 + t6;
t8 = 2*a;
y1 = t7 / t8;
t9 = t1 – t6;
y2 = t9 / t8;
```

```
t1 = -b;
t2 = b*b;
t3 = 4*a;
t3 = t3*c;
a0 = t2 – t3;
v0 = sqrt(a0);
t7 = t1 + v0;
t8 = 2*a;
y1 = t7 / t8;
t9 = t1 – v0;
y2 = t9 / t8;
```

```
lw      t0, b
sub     t1, 0,  t0
mult    t2, t0, t0
lw      t0, a
mult    t3, t0, 4
lw      t4, c
mult    t3, t3, t4
sub     a0, t2, t3
jal     sqrt
add     t7, t1, v0
mult    t8, t0, 2
div     t0, t7, t8;
sw      t0, y1
sub     t9, t1, v0
div     t0, t9, t8
sw      t0, y2
```

```
y1 = (-b + sqrt(b*b - 4*a*c) / (2*a);
y2 = (-b - sqrt(b*b - 4*a*c) / (2*a);
```

```
t1 = -b;            => lw     t0, b
                    => sub    t1, 0,  t0
t2 = b*b;           => mult   t2, t0, t0
t3 = 4*a;           => lw     t0, a
                    => mult   t3, t0, 4
t3 = t3*c;          => lw     t4, c
                    => mult   t3, t3, t4
a0 = t2 – t3;       => sub    a0, t2, t3
v0 = sqrt(a0);      => jal    sqrt
t7 = t1 + v0;       => add    t7, t1, v0
t8 = 2*a;           => mult   t8, t0, 2
y1 = t7 / t8;       => div    t0, t7, t8;
                    => sw     t0, y1
t9 = t1 – v0;       => sub    t9, t1, v0
y2 = t9 / t8;       => div    t0, t9, t8
                    => sw     t0, y2
```

Variables

- Can be held in Registers
  - Temporary variables
  - Internal to expression evaluation
  - Local to a function and no references to them
  - Arguments and return values
- Or in memory
  - Global or static variables (externals)
  - Local variables on the stack
  - Values in the heap

- Memory is usually accessed indirectly through a (special) register
  - stack pointer
  - global pointer
  - heap pointer

Variable examples

```
int ext;

int foo (int n) {
       int loc;
       int A [8];
       struct {int x;
           int y;} point;
       int dyn[] = malloc(10*sizeof(int));
       …
       return (loc+n);
}
```

Conditionals

Human C code

```
if (condition) {true-clause }
else {false_clause }
```

Machine-level C code

```
if (condition) goto Ltrue;
false_clause
goto Ldone;
Ltrue:
true_clause
Ldone:
```

Machine-level Assembly code

```
BR_condition Ltrue
code for false_clause
jmp Ldone
Ltrue:
code for true_clause
Ldone:
```

Jumps and Branches

- Jumps – unconditional control transfers
  - direct or indirect
  - "calls" are a special Jump-and-link
    - » saves the return address
  - computes target address and loads PC
- Branches – conditional control transfers
  - tests a condition and branches if true
  - otherwise falls through sequentially
  - MIPS provides simple conditions on registers
    - » BEQ, BNE, BGZ, …

Loops

Human C code

```
while (condition) {loop body }
```

Machine-level C code

```
Ltop:
if (!condition) goto Ldone;
loop body
goto Ltop;
Ldone:
```

Machine-level Assembly code

```
Ltop:
BR_condition Ltrue
jmp Ldone
Ltrue:
code for loop_body
jmp Ltop
Ldone:
```

Functions (basic)

Human C code

```
int foo (int arg1, int arg2) {
  declarations;
  function_body;
  return val;
}

…
res = foo(param1, param2);
…
```

Machine-level Assembly code

```
foo:
  access arg1 as $a0
  access arg2 as $a1
…
  result in $v0
  jr $ra


$a0 <= param1
$a1 <= param2
jal foo
access $v0
```

MIPS register conventions

Name	Number	Use	Callee must preserve?
$zero	$0	constant 0	N/A
$at	$1	assembler temporary	No
$v0–$v1	$2–$3	returns values	No
$a0–$a3	$4–$7	function arguments	No
$t0–$t7	$8–$15	temporaries	No
$s0–$s7	$16–$23	saved temporaries	Yes
$t8–$t9	$24–$25	temporaries	No
$k0–$k1	$26–$27	reserved for OS kernel	No
$gp	$28	global pointer	Yes
$sp	$29	stack pointer	Yes
$fp	$30	frame pointer	Yes
$ra	$31	return address	N/A

but…

- What if the procedure calls another procedure?
  - $ra will get clobbered by the nested call!
  - How do I save it?
- The function is suppose to save any $s registers it uses. How does it do that?
- The function gets to clobber the $t registers, if the caller is still using some, how does it save them?
- What about the stack pointer?

- On the stack!
  - and a little bit of moving things between registers

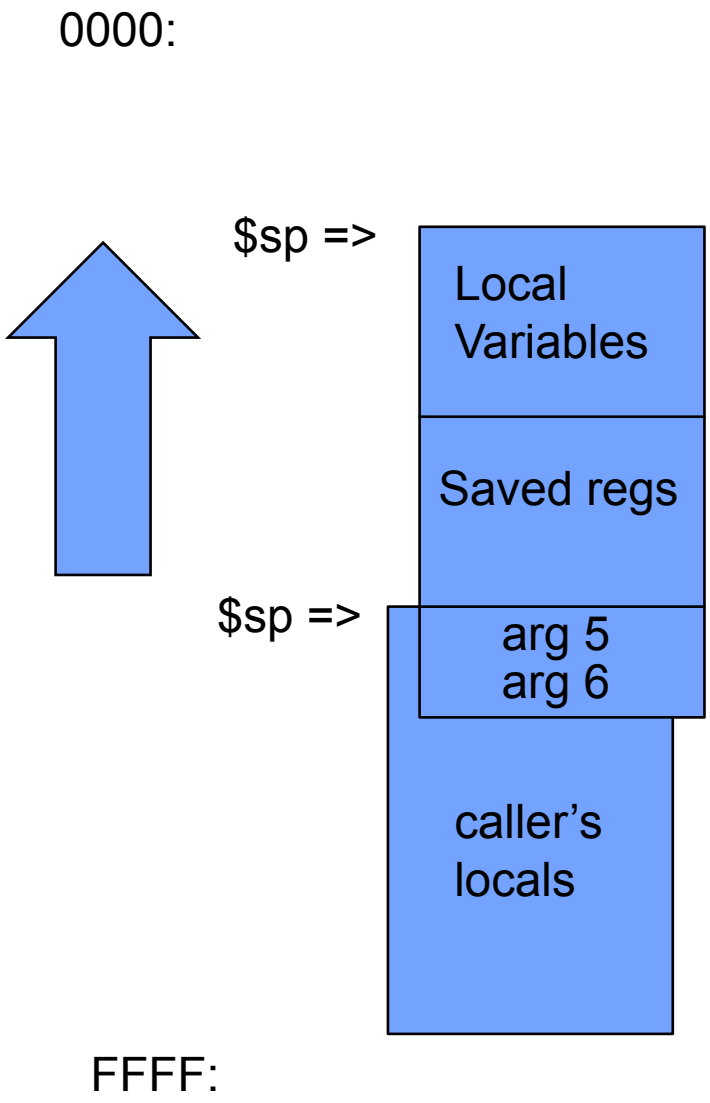
The stack

Calling Conventions:

- **first 4 args in $a0-$a4**
- **results in $v0-$v1**

Hardware:

- **$ra set by jal to following instruction**

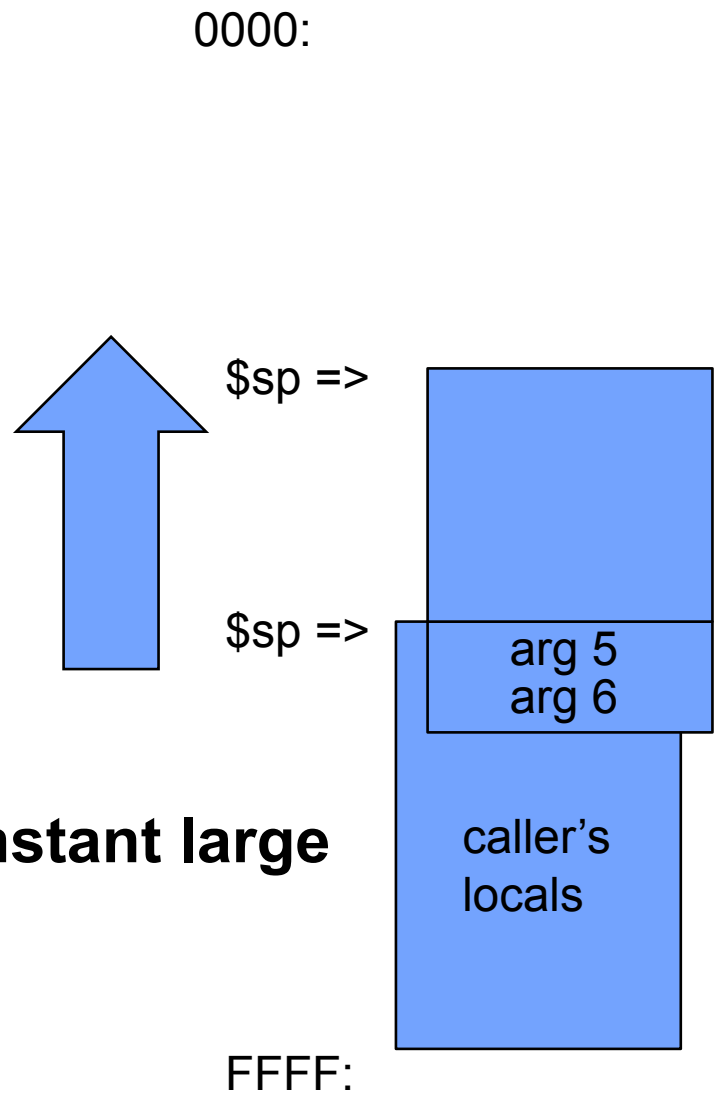


Functions ($ra)

```
main:
…
        jal foo

…
foo:
…
        jr $ra
```

Functions ($sp)

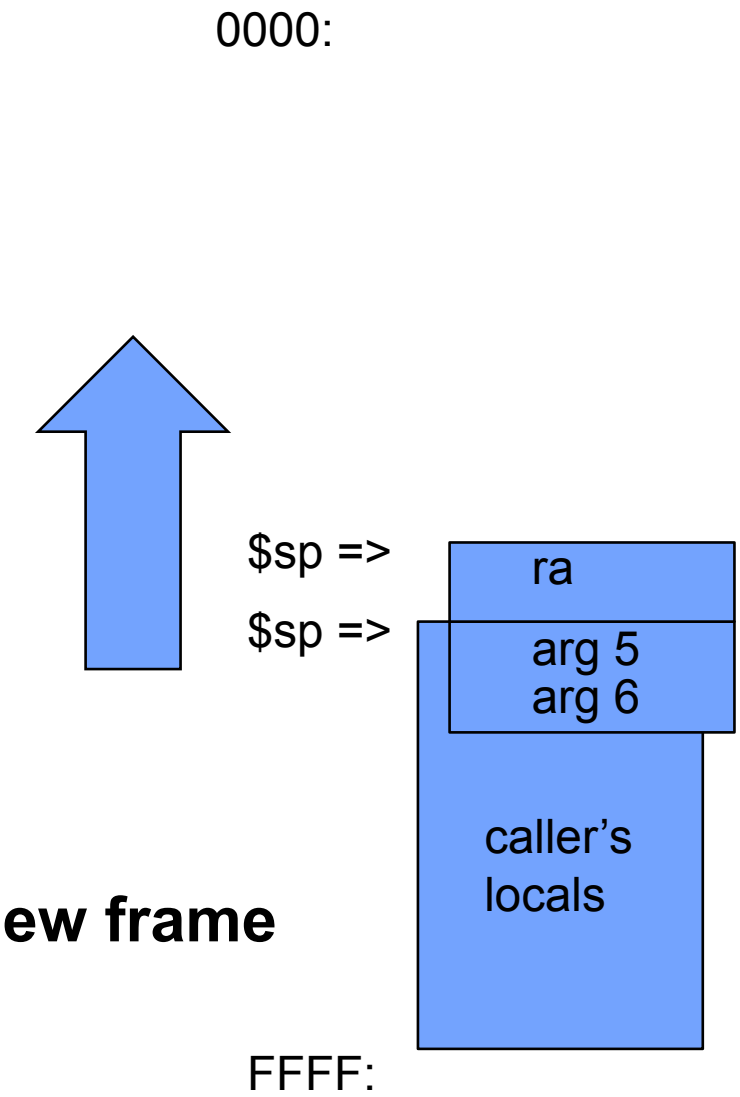
```
main:
…
        jal foo

…
foo:
        subu  $sp, $sp, 32
        …
        addu  $sp, $sp, 32
        jr $ra
```



- **Adjust the stack ptr by a constant large enough to hold the frame**
- **Restore before returning**

Functions ($ra)

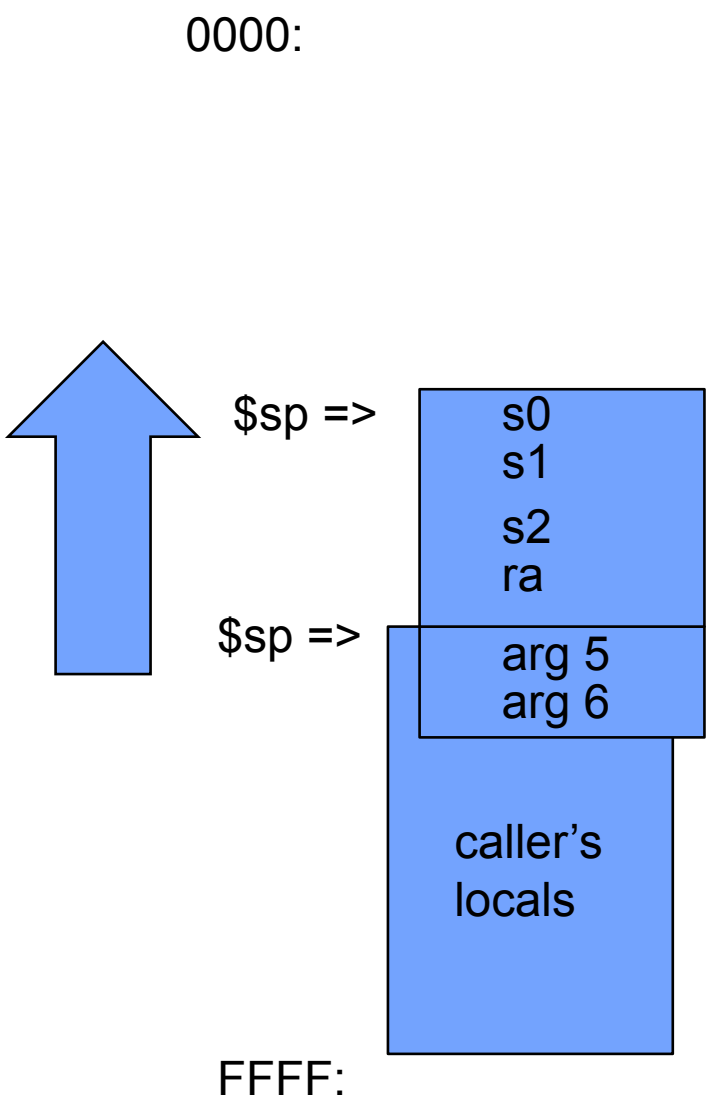
```
main:
…
        jal foo

…
foo:
        subu  $sp, $sp, 4
        sw    $ra, 0($sp)
        …
        lw    $ra, 0($sp)
        addu  $sp, $sp, 4
        jr $ra
```



- **Save return address in the new frame**
- **Restore it before returning**

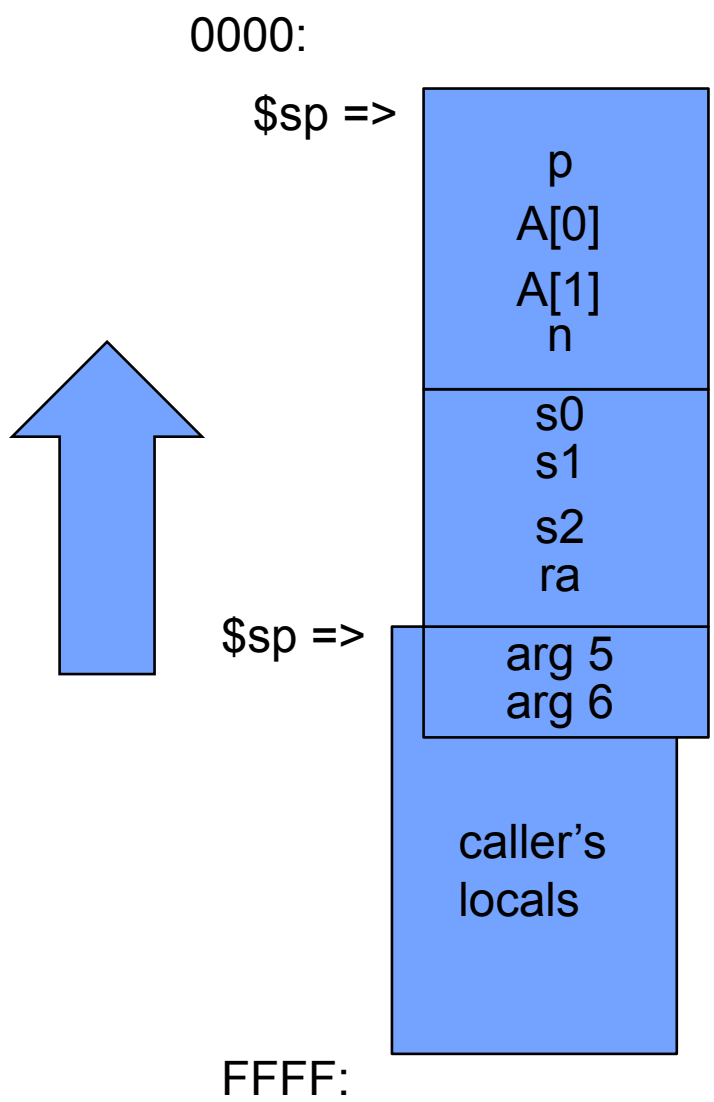
Functions ($s registers)

```
main:
…
        jal foo
…
foo:
        subu  $sp, $sp, 16
        sw    $s0, 0($sp)
        sw    $s1, 4($sp)
        sw    $s2, 8($sp)
        sw    $ra, 12($sp)
        …
        lw    $s0, 0($sp)
        lw    $s1, 4($sp)
        lw    $s2, 8($sp)
        lw    $ra, 12($sp)
        addu  $sp, $sp, 16
        jr $ra
```



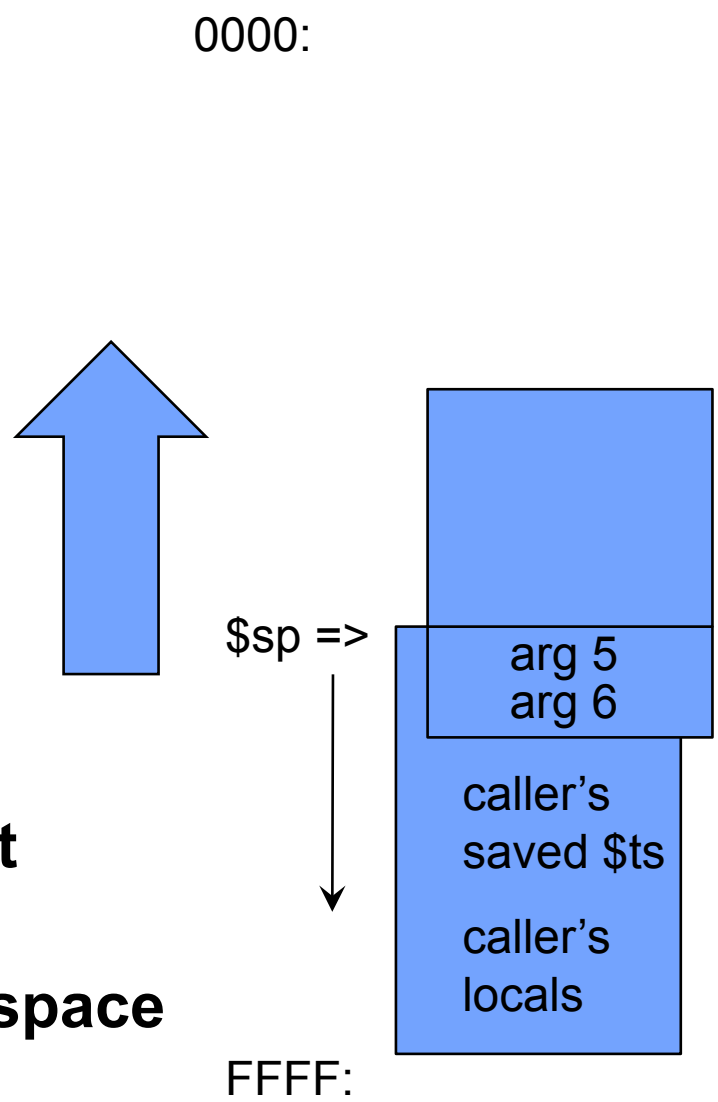
Functions (locals)

```
main:
…
        jal foo

…
foo:
        subu  $sp, $sp, 40
        sw    $s0, 24($sp)
        sw    $s1, 28($sp)
        sw    $s2, 32($sp)
        sw    $ra, 36($sp)
        …
        sw    $xx, (0)$sp
        …
        lw    $s0, 24($sp)
        lw    $s1, 28($sp)
        lw    $s2, 32($sp)
        lw    $ra, 36($sp)
        addu  $sp, $sp, 40
        jr $ra
```



Functions ($t's)

```
main:
        subu  $sp, $sp, 40
        …
        sw    $t1, 16($sp)
        sw    $t3, 20($sp)
        jal foo
        lw    $t1, 16($sp)
        lw    $t3, 20($sp)

…
foo:
        …
        jr $ra
```



- **only save $ts that are "live" at point of call**
- **anticipate need by reserving space in frame on entry**

Summary

- Compiler "expands" language elements into machine operations
  - declarations, sequence, conditional, iteration, function
- Register usage established "by convention"
  - hardware dictates some specific usage ($ra)
  - $sp, $gp, $a0-3, $v0-1, $s0-7, $t0-9
- Calling Convention used systematically
  - stack frame per call
  - save/restore $sp (arithmetically)
  - >4 args "pushed" before call
  - caller saves $t's that it still wants, callee can trash any $t's
  - callee saves $s's that it uses, caller can assume all restored
  - $ra saved/restored if not a leaf proceedure
  - Locals on the stack, discarded when $sp restored
- Enables nesting and recursion