Comp Org Review

Common CPU features:

A. A collection of N general purpose registers, decimal labeled R0, R1, R10... Some architectures permit "double word" sized values to be stored across two sequential registers, so they have D0, D2, D4, ...

B. A PC register holding the address of **the next instruction** to execute, and associated operations that manipulate it such as `jump`, `call`, `return`, ...

C. An SP register "stack pointer" and associated operations such as `push R2`, `dpop D3` (double word pop), ...

Comp Org Review

Common CPU features continued:

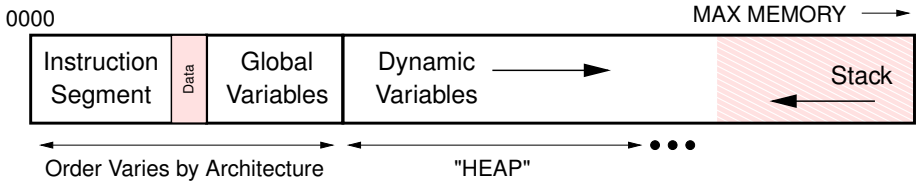
D. A return address register (RA) that stores the return-to instruction address for a function (procedure, subprogram) call,

E. A frame pointer register (FP) that stores a function's **activation record** or **frame** (more on this later)

F. A method for addressing memory locations using `(Register,Offset)` notation, where the `Register` is any CPU register and the `Offset` is usually a literal integer value (some architectures support another register), eg:
`LOAD R0,(SP,4)` (load machine word at 4 bytes past the stack pointer into R0),
`STORE (R11,R2),R3` (if C `array` address is stored in `R11` and variables `i` and `v` are in `R2` and `R3`, this is C code `array[i]=v;`)

Very small, basic CPUs may not have dedicated RA or FP registers, in which case the compiler designer designates general purpose registers to the task(s).

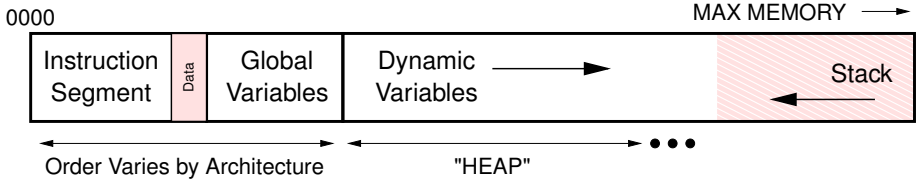
Memory



1. (Our course convention) the SP starts at MAX MEMORY and grows downward. So pushing a value onto the stack decreases the value of SP. The last machine word pushed to the stack is at `SP[0]`.
2. At the base of memory, we have (in some order) **program instructions**, **constant data** and **global variables**. CZAR ordering will likely be different, but we'll see `:)`.
3. The memory after this begins the heap which is used for dynamically allocated storage (we won't cover this topic in detail).

Beware: the text figures for stacks and activation frames can bounce around between stacks that grow down or up. The figures are correct — you just have to look at them twice and be careful of what you're reading.

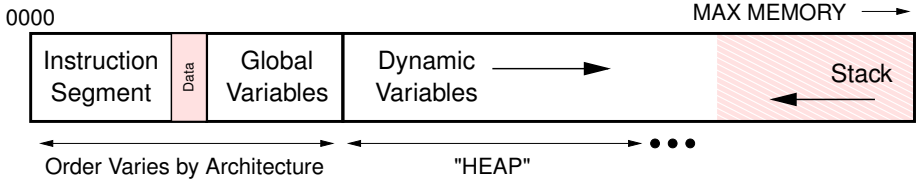
Memory



1. (Our course convention) the SP starts at MAX MEMORY and grows downward. So pushing a value onto the stack decreases the value of SP. The last machine word pushed to the stack is at `SP[0]`.
2. At the base of memory, we have (in some order) **program instructions**, **constant data** and **global variables**. CZAR ordering will likely be different, but we'll see `:)`.
3. The memory after this begins the heap which is used for dynamically allocated storage (we won't cover this topic in detail).

Where are local variables stored?

Memory



1. (Our course convention) the SP starts at MAX MEMORY and grows downward. So pushing a value onto the stack decreases the value of SP. The last machine word pushed to the stack is at `SP[0]`.
2. At the base of memory, we have (in some order) **program instructions**, **constant data** and **global variables**. CZAR ordering will likely be different, but we'll see `:)`.
3. The memory after this begins the heap which is used for dynamically allocated storage (we won't cover this topic in detail).

Where are local variables stored? On the stack.

Procedure, Function and Method Calls

Implementing “recursive capable subprograms” in a compiler is essentially the same whether the invoked logic is a procedure (a function without a return value), function, or object (class) method call.

Clearly, permitting recursion within a single function or a set of functions, requires distinct memory locations for each function instance’s local variables.

The local variables, as well as other **control information** required for stack management are (typically) **allocated in a dynamic fashion on the stack**. Common terms for these function invocation specific memory chunks: **activation record**, **frame**, or **stack frame**.

Accounting Steps for Basic Stack Frames

1. Using a specialized **visitor pass** function on the program AST,
   i. Collect a list of all local variables required per function as well as their type
   ii. Accounting for machine architecture alignment requirements, store each variable's offset in the function's symbol table entry.
   (Word sized variables usually need to be "word aligned" in memory, likewise for double word sized variables.)
2. Save the total storage required for local variables, increased to the next word-aligned boundary if necessary, in the function's symbol table entry.

(The same visitor pass logic should probably collect similar information for program **constant data** as well as **global variables**, but we are focusing on activation record generation right now.)

Creating *Basic* Stack Frames for Function Calls

During the code generation phase of compilation, after the parameters for a function call have been calculated, when we are ready to collect the function call result . . .

We create the **stack frame** for the callee:

1. . . .
2. Push the **control information onto the stack**:
   i. the current (**caller**) function's own stack frame pointer (FP),
   ii. the current (**caller**) function's RA value,
   iii. the stack space for the **callee** function's return value,
3. Pass arguments **to the callee**. Architectures typically pass the first p parameters in dedicated registers, remaining parameters on the stack. $p = 0$ means all parameters are passed on the stack).
4. Now using the **symbol table stored data for the callee**, push the SP sufficiently for the function's local variable storage.
5. Emit a **call** or **jump** instruction to the instruction code base address of the callee function.
6. . . .

Example 1 - C like language

```
bool test( float a, int b )
{
    return a < b;
}
int f()
{
    float big = 0;
    int small = 100;
    int count = 0;
    bool b = test( big, small);
    while ( b ) {
        big = big * 1.1;
        int diff = int(big)
        small = small - diff;
        count = count + 1;
        b = test(big, small);
    }
    return count;
}
int main() { f(); }
```

Locals Collected for `test`

scope	id	type	size	offset	"padding"
1	a	float	8	0	0
1	b	int	4	8	0
total size 12					

It looks like the machine has 4B words, and double words for `float`.

Example 1 - C like language

```
bool test( float a, int b )
{
    return a < b;
}
int f()
{
    float big = 0;
    int small = 100;
    int count = 0;
    bool b = test( big, small);
    while ( b ) {
        big = big * 1.1;
        int diff = int(big)
        small = small - diff;
        count = count + 1;
        b = test(big, small);
    }
    return count;
}
int main() { f(); }
```

Locals Collected for `f`

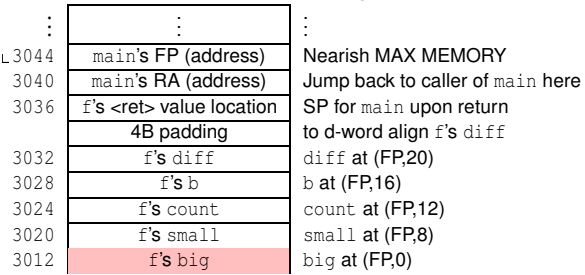
scope	id	type	size	offset	“padding”
2	big	float	8	0	0
2	small	int	4	8	0
2	count	int	4	12	0
2	b	bool	1	16	3
3	diff	int	4	20	0
local’s size 24					

It looks like the compiler is using 1B for `bools`.

Example 1 - C like language — Stack Setup for `main`→`f` Call

`f`'s local data in symbol table

scope	id	type	size	offset	"padding"
2	big	float	8	0	0
2	small	int	4	8	0
2	count	int	4	12	0
2	b	bool	1	16	3
3	diff	int	4	20	0
local's size 24					



⌞3044 means `main`'s FP address is at 3044, `f`'s FP has the value 3012.

The address label of a cell refers to the lower visual edge or border.

Example 1 - C like language — Stack Setup for `main`→`f` Call

`f`'s local data in symbol table

scope	id	type	size	offset	"padding"
2	big	float	8	0	0
2	small	int	4	8	0
2	count	int	4	12	0
2	b	bool	1	16	3
3	diff	int	4	20	0
local's size 24					

addr	contents	description
⋮	⋮	⋮
3044	`main`'s FP (address)	Nearish MAX MEMORY
3040	`main`'s RA (address)	Jump back to caller of `main` here
3036	`f`'s <ret> value location	SP for `main` upon return
	4B padding	to d-word align `f`'s `diff`
3032	`f`'s `diff`	`diff` at `f`'s (FP,20)
3028	`f`'s `b`	`b` at `f`'s (FP,16)
3024	`f`'s `count`	`count` at `f`'s (FP,12)
3020	`f`'s `small`	`small` at `f`'s (FP,8)
3012	`f`'s `big`	`big` at `f`'s (FP,0)

$(3036 - 8) \not| 8$ (not divisible by), so 4B of padding is required for the 8 byte float `diff`.

Example 1 - C like language

`f`'s local data in symbol table

scope	id	type	size	offset	"padding"
2	big	float	8	0	0
2	small	int	4	8	0
2	count	int	4	12	0
2	b	bool	1	16	3
3	diff	int	4	20	0
local's size 24					

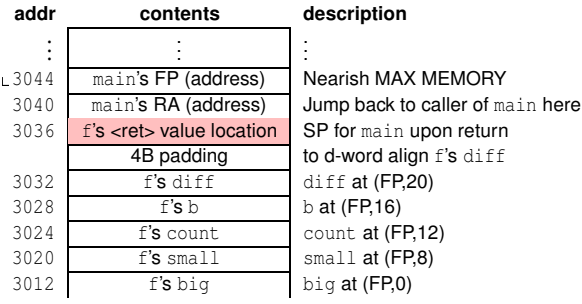
addr	contents	description
⋮	⋮	⋮
3044	`main`'s FP (address)	Nearish MAX MEMORY
3040	`main`'s RA (address)	Jump back to caller of `main` here
3036	`f`'s <ret> value location	SP for `main` upon return
	4B padding	to d-word align `f`'s `diff`
3032	`f`'s `diff`	`diff` at (FP,20)
3028	`f`'s `b`	`b` at (FP,16)
3024	`f`'s `count`	`count` at (FP,12)
3020	`f`'s `small`	`small` at (FP,8)
3012	`f`'s `big`	`big` at (FP,0)

`f` has no parameters, they would have gone between `f`'s <ret> location and `f`'s `diff` — perhaps eliminating the need for padding

Example 1 - C like language

`f`'s local data in symbol table

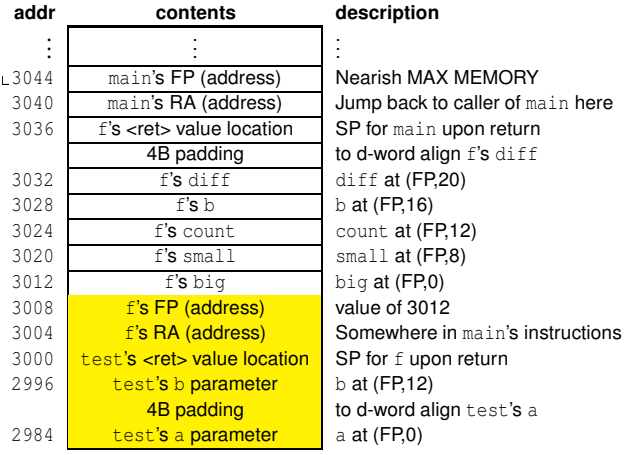
scope	id	type	size	offset	"padding"
2	big	float	8	0	0
2	small	int	4	8	0
2	count	int	4	12	0
2	b	bool	1	16	3
3	diff	int	4	20	0
local's size 24					



This architecture puts return values on the stack, some use specific registers for machine word return values.

Example 1 - C like language — Stack Setup for `f`→`test` Call

```
bool test( float a, int b )
{
    return a < b;
}
int f()
{
    float big = 0;
    int small = 100;
    int count = 0;
    bool b = test( big, small);
    while ( b ) {
        big = big * 1.1;
        int diff = int(big)
        small = small - diff;
        count = count + 1;
        b = test(big, small);
    }
    return count;
}
int main() { f(); }
```



Example 1 - C like language — Stack Setup for `f`→`test` Call

`f`'s local data in symbol table

scope	id	type	size	offset	"padding"
2	big	float	8	0	0
2	small	int	4	8	0
2	count	int	4	12	0
2	b	bool	1	16	3
3	diff	int	4	20	0
local's size 24					

`test`'s local data in symbol table

scope	id	type	size	offset	"padding"
1	a	float	8	0	0
1	b	int	4	8	0
total size 12					

addr	contents	description
⋮	⋮	⋮
⌞3044	`main`'s FP (address)	Nearish MAX MEMORY
3040	`main`'s RA (address)	Jump back to caller of `main` here
3036	`f`'s <ret> value location	SP for `main` upon return
	4B padding	to d-word align `f`'s `diff`
3032	`f`'s `diff`	`diff` at (FP,20)
3028	`f`'s `b`	`b` at (FP,16)
3024	`f`'s `count`	`count` at (FP,12)
3020	`f`'s `small`	`small` at (FP,8)
3012	`f`'s `big`	`big` at (FP,0)
3008	`f`'s FP (address)	value of 3012
3004	`f`'s RA (address)	Somewhere in `main`'s instructions
3000	`test`'s <ret> value location	SP for `f` upon return
2996	`test`'s `b` parameter	`b` at (FP,12)
	4B padding	to d-word align `test`'s `a`
2984	`test`'s `a` parameter	`a` at (FP,0)

This architecture puts all arguments on the stack, some pass the first p arguments in registers.

Example 1 - C like language — Stack Setup for `f`→`test` Call

`f`'s local data in symbol table

scope	id	type	size	offset	"padding"
2	big	float	8	0	0
2	small	int	4	8	0
2	count	int	4	12	0
2	b	bool	1	16	3
3	diff	int	4	20	0
local's size 24					

`test`'s local data in symbol table

scope	id	type	size	offset	"padding"
1	a	float	8	0	0
1	b	int	4	8	0
total size 12					

addr	contents	description
⋮	⋮	⋮
⌞3044	`main`'s FP (address)	Nearish MAX MEMORY
3040	`main`'s RA (address)	Jump back to caller of `main` here
3036	`f`'s <ret> value location	SP for `main` upon return
	4B padding	to d-word align `f`'s `diff`
3032	`f`'s `diff`	`diff` at (FP,20)
3028	`f`'s `b`	`b` at (FP,16)
3024	`f`'s `count`	`count` at (FP,12)
3020	`f`'s `small`	`small` at (FP,8)
3012	`f`'s `big`	`big` at (FP,0)
3008	`f`'s FP (address)	value of 3012
3004	`f`'s RA (address)	Somewhere in `main`'s instructions
3000	`test`'s <ret> value location	SP for `f` upon return
2996	`test`'s `b` parameter	`b` at (FP,12)
	4B padding	to d-word align `test`'s `a`
2984	`test`'s `a` parameter	`a` at (FP,0)

`test` has no local variables, so its FP value is 2984

Example 1 - C like language — Stack Setup for `f`→`test` Call

- There are straight-forward algorithms for “compacting” local variables that consume varying amounts of space — **they don’t have to be placed on the stack in order of their declaration**.
  (Collect like types sequentially in the stack, for instance in the previous example you could fit four 1B `bools` into the space of an `int`...).

- The ordering of control information, parameters, and local variables varies from architecture to architecture. As long as the caller and the callee use the same organization — everything works out.
  Don’t be freaked out when you see other orderings in our text, other texts and of course the web.

Still a Basic Frame

The example's stack is not precisely correct, there are several details we've glossed over: (see the last slide) . . .

But this is a good (re)introduction to stack management, which will hopefully make the LGA reading more accessible.

Next up: register counting and code generation.

Remaining Topics in LGA Reading

1. Why two registers? Why can't SP and FP be treated the same?
2. How will a function's generated code access local variables?
3. What is the "dynamic link" in the activation record?
4. What is the difference between **procedure-level-frame-allocation** and **block-level-frame-allocation**, what are these techniques used for?
5. What are **callee** versus **caller** saved registers?