The Stack & Procedures II
CSE 351 Winter 2021

Instructor:
Mark Wyse

Teaching Assistants:
Kyrie Dowling
Catherine Guevara
Ian Hsiao
Jim Limprasert
Armin Magness
Allie Pfleger
Cosmo Wang
Ronald Widjaja

http://xkcd.com/1270/
Administrivia

❖ Lab 2 – now due Monday (2/8)
  ▪ Since you are submitting a text file (defuser.txt), there won’t be any Gradescope autograder output this time
  ▪ Extra credit (bonus) needs to be submitted to the extra credit assignment

❖ hw11 due Wednesday, hw12 due Friday

❖ Section worksheets
  ▪ will now be due on the Monday following section
Reading Review

❖ Terminology:
  ▪ Stack frame: return address, saved registers, local variables, argument build
  ▪ Register saving conventions: callee-saved and caller-saved

❖ Questions from the Reading?
x86-64/Linux Stack Frame

- **Caller’s Stack Frame**
  - Extra arguments (if > 6 args) for this call

- **Current/Callee Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Old frame pointer (optional)
  - Saved register context (when reusing registers)
  - Local variables
    (If can’t be kept in registers)
  - “Argument build” area
    (If callee needs to call another function - parameters for function about to call, if needed)
Example: increment

```c
long increment(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

increment:
- `movq (%rdi), %rax`  
- `addq %rax, %rsi`  
- `movq %rsi, (%rdi)`  
- `ret`

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st arg (p)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd arg (val), y</td>
</tr>
<tr>
<td>%rax</td>
<td>x, return value</td>
</tr>
</tbody>
</table>
Procedure Call Example (initial state)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

- Return address on stack is the address of instruction immediately following the call to “call_incr”
  - Shown here as `main`, but could be anything)
  - Pushed onto stack by `call call_incr`

Initial Stack Structure

- Return addr `<main+8>`
Procedure Call Example (step 1)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

Call structure:
- **Allocate space for local vars**
  - Setup space for local variables
    - Only `v1` needs space on the stack
  - Compiler allocated extra space
    - Often does this for a variety of reasons, including alignment

Stack Structure:
- Return addr `<main+8>`
  - `351` Unused
  - `old %rsp` ← `%rsp+8`
  - ← `%rsp`
Procedure Call Example (step 2)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

### Stack Structure

<table>
<thead>
<tr>
<th>Return addr &lt;main+8&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>351</td>
</tr>
<tr>
<td>Unused</td>
</tr>
</tbody>
</table>

Set up parameters for call to increment

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>100</td>
</tr>
</tbody>
</table>

Aside: movl is used because 100 is a small positive value that fits in 32 bits. High order bits of rsi get set to zero automatically. It takes one less byte to encode a movl than a movq.
Procedure Call Example (step 3)

long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}

call_incr:
    subq $16, %rsp
    movq $351, 8(%rsp)
    movl $100, %esi
    leaq 8(%rsp), %rdi
    call increment
    addq 8(%rsp), %rax
    addq $16, %rsp
    ret

Stack Structure

increment:
    movq (%rdi), %rax
    addq %rax, %rsi
    movq %rsi, (%rdi)
    ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>100</td>
</tr>
<tr>
<td>%rax</td>
<td></td>
</tr>
</tbody>
</table>

State while inside `increment`
- **Return address** on top of stack is address of the addq instruction immediately following call to `increment`

Frames can be different sizes!
Procedure Call Example (step 4)

```c
long call_incr() {
  long v1 = 351;
  long v2 = increment(&v1, 100);
  return v1 + v2;
}
```

**Stack Structure**

<table>
<thead>
<tr>
<th>Return addr</th>
<th>451</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unused</td>
<td></td>
</tr>
</tbody>
</table>

**Return addr** `<call_incr+?>`

- **State while inside increment**
  - *After* code in body has been executed

**increment:**

```asm
movq (%rdi), %rax  # x = *p
addq %rax, %rsi   # y = x + 100
movq %rsi, (%rdi) # *p = y
ret
```

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>351</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 5)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

- After returning from call to `increment`
  - Registers and memory have been modified and return address has been popped off stack

---

**Stack Structure**

- Return addr `<main+8>`
- `451` → `%rsp+8`
- `Unused` → `%rsp`

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>&amp;v1</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>451</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>351</td>
</tr>
</tbody>
</table>
**Procedure Call Example** *(step 6)*

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

**Stack Structure**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>451+351</td>
</tr>
</tbody>
</table>

**call_incr:**
- `subq $16, %rsp`
- `movq $351, 8(%rsp)`
- `movl $100, %esi`
- `leaq 8(%rsp), %rdi`
- `call increment`  
- `addq 8(%rsp), %rax`  
- `addq $16, %rsp`
- `ret`

- **Return addr**: `<main+8>`
- **Unused**: `←%rsp+8`, `←%rsp`

- **Update** `%rax` to contain `v1+v2`
Procedure Call Example (step 7)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

**Stack Structure**

```
Return addr <main+8>
451
Unused
```

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 8)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

- State just before returning from call to `call_incr`

- Stack Structure:

  - Return addr `<main+8>`
  - `%rsp`:

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rdi</code></td>
<td>&amp;v1</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>451</td>
</tr>
<tr>
<td><code>%rax</code></td>
<td>802</td>
</tr>
</tbody>
</table>
Procedure Call Example (step 9)

```c
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

- **State immediately after** returning from call to `call_incr`
  - Return addr has been popped off stack
  - Control has returned to the instruction immediately following the call to `call_incr` (not shown here)

**Final Stack Structure**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>&amp;v1</td>
</tr>
<tr>
<td>%rsi</td>
<td>451</td>
</tr>
<tr>
<td>%rax</td>
<td>802</td>
</tr>
</tbody>
</table>
Polling Question

Answer the following questions about when `main()` is run (assume `x` and `y` stored on the Stack):

```c
int main() {
    int i, x = 0;
    for(i=0;i<3;i++)
        x = randSum(x);
    printf("x = %d\n",x);
    return 0;
}
```

- **Higher/larger address**: `x` or `y`?
- **How many total stack frames are created?**
- **What is the maximum depth (# of frames) of the Stack?**

**Vote only on 3rd question on Ed Lessons**

A. 1  B. 2  C. 3  D. 4
Review Question

❖ In the following function, which instruction(s) pertain to the local variables and saved registers portions of its stack frame?

```
call_incr2:
  1 pushq %rbx  # save %rbx
  2 subq $16, %rsp
  3 movq %rdi, %rbx
  4 movq $351, 8(%rsp)  # 351 to stack
  5 movl $100, %esi
  6 leaq 8(%rsp), %rdi
  7 call increment
  8 addq %rbx, %rax  # ret of increment
  9 addq $16, %rsp  # deallocate stack
 10 popq %rbx  # restore %rbx
 11 ret
```

local variables: 2, 4, 6, 9

saved registers: 1, 10
Procedures

❖ Stack Structure
❖ Calling Conventions
  ▪ Passing control
  ▪ Passing data
  ▪ Managing local data
❖ Register Saving Conventions
❖ Illustration of Recursion
Register Saving Conventions

- When procedure `whoa` calls `who`:
  - `whoa` is the caller
  - `who` is the callee

- Can registers be used for temporary storage?

  ```
  whoa:
  ... 
  movq $15213, %rdx
  call who
  addq %rdx, %rax
  ... 
  ret
  
  who:
  ... 
  subq $18213, %rdx
  ... 
  ret
  ```

  - No! Contents of register `%rdx` overwritten by `who`!
  - This could be trouble – something should be done. Either:
    - `Caller` should save `%rdx` before the call (and restore it after the call)
    - `Callee` should save `%rdx` before using it (and restore it before returning)
Register Saving Conventions

❖ “Caller-saved” registers

- It is the caller’s responsibility to save any important data in these registers before calling another procedure (i.e., the callee can freely change data in these registers)
- Caller saves values in its stack frame before calling Callee, then restores values after the call

❖ “Callee-saved” registers

- It is the callee’s responsibility to save any data in these registers before using the registers (i.e., the caller assumes the data will be the same across the callee procedure call)
- Callee saves values in its stack frame before using, then restores them before returning to caller
Silly Register Convention Analogy

1) Parents (\textit{caller}) leave for the weekend and give the keys to the house to their child (\textit{callee})
   - Being suspicious, they put away/hid the valuables (\textit{caller-saved}) before leaving
   - Warn child to leave the bedrooms untouched: “These rooms better look the same when we return!”

2) Child decides to throw a wild party (\textit{computation}), spanning the entire house
   - To avoid being disowned, child moves all of the stuff from the bedrooms to the backyard shed (\textit{callee-saved}) before the guests trash the house
   - Child cleans up house after the party and moves stuff back to bedrooms

3) Parents return home and are satisfied with the state of the house
   - Move valuables back and continue with their lives
x86-64 Linux Register Usage, part 1

❖ \%rax
  - Return value
  - Also **caller**-saved & restored
  - Can be modified by procedure

❖ \%rdi, \ldots, \%r9
  - Arguments
  - Also **caller**-saved & restored
  - Can be modified by procedure

❖ \%r10, \%r11
  - **Caller**-saved & restored
  - Can be modified by procedure
x86-64 Linux Register Usage, part 2

- **%rbx, %r12, %r13, %r14**
  - **Callee**-saved
  - **Callee** must save & restore

- **%rbp**
  - **Callee**-saved
  - **Callee** must save & restore
  - May be used as frame pointer
  - Can mix & match

- **%rsp**
  - Special form of **callee** save
  - Restored to original value upon exit from procedure
### x86-64 64-bit Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Saved by</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%rax</code></td>
<td>Return value</td>
<td>Caller saved</td>
</tr>
<tr>
<td><code>%rbx</code></td>
<td></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%rcx</code></td>
<td>Argument #4</td>
<td>Caller saved</td>
</tr>
<tr>
<td><code>%rdx</code></td>
<td>Argument #3</td>
<td>Caller saved</td>
</tr>
<tr>
<td><code>%rsi</code></td>
<td>Argument #2</td>
<td>Caller saved</td>
</tr>
<tr>
<td><code>%rdi</code></td>
<td>Argument #1</td>
<td>Caller saved</td>
</tr>
<tr>
<td><code>%rsp</code></td>
<td>Stack pointer</td>
<td>Callee</td>
</tr>
<tr>
<td><code>%rbp</code></td>
<td></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%r8</code></td>
<td>Argument #5</td>
<td>Caller saved</td>
</tr>
<tr>
<td><code>%r9</code></td>
<td>Argument #6</td>
<td>Caller saved</td>
</tr>
<tr>
<td><code>%r10</code></td>
<td></td>
<td>Caller saved</td>
</tr>
<tr>
<td><code>%r11</code></td>
<td></td>
<td>Caller saved</td>
</tr>
<tr>
<td><code>%r12</code></td>
<td></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%r13</code></td>
<td></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%r14</code></td>
<td></td>
<td>Callee saved</td>
</tr>
<tr>
<td><code>%r15</code></td>
<td></td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
Callee-Saved Example (step 1)

```c
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

Initial Stack Structure

```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

Resulting Stack Structure

```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```
Callee-Saved Example (step 2)

```c
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

call_incr2:

1. `pushq %rbx`
2. `subq $16, %rsp`
3. `movq %rdi, %rbx`
4. `movq $351, 8(%rsp)`
5. `movl $100, %esi`
6. `leaq 8(%rsp), %rdi`
7. `call increment`
8. `addq %rbx, %rax`
9. `addq $16, %rsp`
10. `popq %rbx`
11. `ret`

Memory Stack Structure

- Rtn address
- Saved %rbx
- 351
- Unused

Registers

- %rbx
- %rdi
- %esi
- %rax

Pre-return Stack Structure

- Rtn address
- %rsp

Stack discipline:
- Add/sub push/pull must be symmetric within procedure
Why Caller and Callee Saved?

- We want one calling convention to simply separate implementation details between caller and callee.

- In general, neither caller-save nor callee-save is “best”:
  - If caller isn’t using a register, caller-save is better.
  - If callee doesn’t need a register, callee-save is better.
  - If “do need to save”, callee-save generally makes smaller programs.
    • Functions are called from multiple places.

- So... “some of each” and compiler tries to “pick registers” that minimize amount of saving/restoring.
Register Conventions Summary

- **Caller**-saved register values need to be pushed onto the stack before making a procedure call *only if the Caller needs that value later*
  - **Callee** may change those register values

- **Callee**-saved register values need to be pushed onto the stack *only if the Callee intends to use those registers*
  - **Caller** expects unchanged values in those registers

- Don’t forget to restore/pop the values later!
  - 1 push -> 1 pop
Procedures

❖ Stack Structure
❖ Calling Conventions
  ▪ Passing control
  ▪ Passing data
  ▪ Managing local data
❖ Register Saving Conventions
❖ Illustration of Recursion
Recursive Function

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

Compiler Explorer:

- [Compiler Explorer](https://godbolt.org/z/naP4ax)
  - Compiled with `-O1` for brevity instead of `-Og`
  - Try `-O2` instead!

```
movl $0, %eax
testq %rdi, %rdi
jne .L8
rep ret
.L8:
pushq %rbx
movq %rdi, %rbx
shrq %rdi
call pcount_r
andl $1, %ebx
addq %rbx, %rax
popq %rbx
ret
```
Recursive Function: Base Case

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

Trick because some AMD hardware doesn’t like jumping to `ret`
Recursive Function: **Callee Register Save**

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x</td>
<td>Argument</td>
</tr>
</tbody>
</table>

The Stack

- Need original value of \( x \) *after* recursive call to `pcount_r`.
- “Save” by putting in \( \%rbx \) (*callee saved*), but need to save old value of \( \%rbx \) before you change it.

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    jne .L8
    rep ret
.L8:
    pushq %rbx
    movq %rdi, %rbx
    shrq %rdi
    call pcount_r
    andl $1, %ebx
    addq %rbx, %rax
    popq %rbx
    ret
```
Recursive Function: Call Setup

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>x (new)</td>
<td>Argument</td>
</tr>
<tr>
<td>%rbx</td>
<td>x (old)</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

### The Stack

- `%rsp` →
- `saved %rbx`
- `rtn <main+?>`
- `...`

### pcount_r:

```
movl $0, %eax
testq %rdi, %rdi
jne .L8
rep ret

.L8:
pushq %rbx
movq %rdi, %rbx
shrq %rdi
call pcount_r
andl $1, %ebx
addq %rbx, %rax
popq %rbx
ret
```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}

The Stack

x = 0b10
rtn <main+?>
saved %rbx
rtn <pcount_r+22>

%rsp → %rax

Register | Use(s) | Type
---|---|---
%rax | Recursive call return value | Return value
%rbx | x (old) | Callee saved

pcount_r:
movl $0, %eax
testq %rdi, %rdi
jne .L8
rep ret
.L8:
pushq %rbx
movq %rdi, %rbx
shrq %rdi
call pcount_r
andl $1, %ebx
addq %rbx, %rax
popq %rbx
ret
Recursive Function: Result

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

### The Stack

<table>
<thead>
<tr>
<th>%rsp</th>
<th>saved %rbx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rtn &lt;main+?&gt;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>x&amp;1</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

### pcount_r:

```
movl $0, %eax
movq %rdi, %rbx
shrq %rdi
call pcount_r
andl $1, %ebx
addq %rbx, %rax
popq %rbx
ret
```

The `pcount_r` function is recursively calculating the set bit count of a number. The stack frame includes the return address and the callee-saved register %rbx. The disassembly shows the steps involved in calculating the set bit count using the POPCNT (Pop Count) instruction.
Recursive Function: Completion

```c
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Previous %rbx value</td>
<td>Callee restored</td>
</tr>
</tbody>
</table>

The Stack

```
%rsp →

rtn <main+><?

saved %rbx
```

pcount_r:

```
movl $0, %eax
```

```
testq %rdi, %rdi
```

```
jne .L8
```

```
rep ret .L8:
```

```
pushq %rbx
```

```
movq %rdi, %rbx
```

```
shrq %rdi
```

```
call pcount_r
```

```
andl $1, %ebx
```

```
addq %rbx, %rax
```

```
popq %rbx
```

```
ret
```


Observations About Recursion

❖ Works without any special consideration
  ▪ Stack frames mean that each function call has private storage
    • Saved registers & local variables
    • Saved return address
  ▪ Register saving conventions prevent one function call from corrupting another’s data
    • Unless the code explicitly does so (e.g. buffer overflow)
  ▪ Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
      • Last-In, First-Out (LIFO)
❖ Also works for mutual recursion (P calls Q; Q calls P)
x86-64 Stack Frames

❖ Many x86-64 procedures have a minimal stack frame
  ▪ Only return address is pushed onto the stack when procedure is called

❖ A procedure *needs* to grow its stack frame when it:
  ▪ Has too many local variables to hold in caller-saved registers
  ▪ Has local variables that are arrays or structs
  ▪ Uses & to compute the address of a local variable
  ▪ Calls another function that takes more than six arguments
  ▪ Is using caller-saved registers and then calls a procedure
  ▪ Modifies/uses callee-saved registers
x86-64 Procedure Summary

❖ Important Points

▪ Procedures are a combination of instructions and conventions
  • Conventions prevent functions from disrupting each other
▪ Stack is the right data structure for procedure call/return
  • If P calls Q, then Q returns before P
▪ Recursion handled by normal calling conventions

❖ Heavy use of registers

▪ Faster than using memory
▪ Use limited by data size and conventions

❖ Minimize use of the Stack