x86 assembly

CS449 Spring 2016

CISC vs. RISC

CISC *[Complex instruction set Computing]* - larger, more feature-rich instruction set (more operations, addressing modes, etc.). slower clock speeds. fewer general purpose registers. Examples: x86 variants

- **E.g.**: F2XM1 – Compute 2x-1
  - Computes the exponential value of 2 to the power of the source operand minus 1. The source operand is located in register ST(0) and the result is also stored in ST(0). The value of the source operand must lie in the range -1.0 to +1.0. If the source value is outside this range, the result is undefined.

RISC *[Reduced instruction set Computing]* - smaller, simpler instruction set. faster clock speeds. more general purpose registers. Examples: MIPS, ARM, PIC, Itanium, PowerPC

Modern processors are pretty much all RISC. Even CISC instruction sets (x86-64) are translated to RISC microcode on chip prior to execution.

32-Bit General Purpose Registers

- EAX – Accumulator
- EBX – Base
- ECX – Counter
- EDX – Data

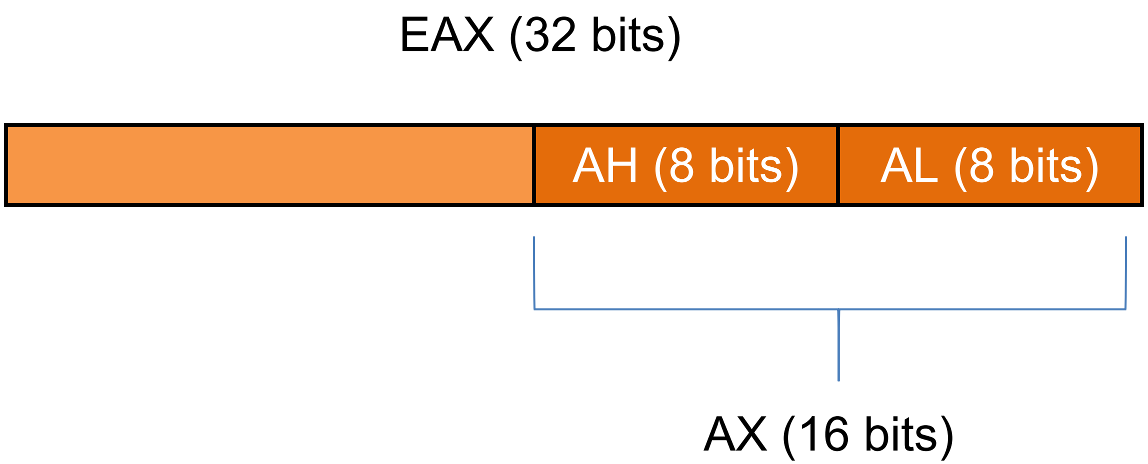
- ESI – String Source
- EDI – String Destination

Other 32-Bit Registers

- EIP – Instruction Pointer
- ESP – Stack Pointer
- EBP – Base or Frame Pointer

- EFLAGS – Flag register

Register Subfields



AT&T Syntax

- `gcc` and `gas` use AT&T syntax:
  - Opcode appended by type
    - b – byte (1 byte)
    - w – word (2 bytes)
    - l – long (4 bytes)
    - q – quad (8 bytes)
  - First operand is source
  - Second operand is destination
  - Memory dereferences are denoted by ( )

AT&T Hello World

```c
//C program
int main()
{
     puts("Hello world!");
     return 0;
}
```

```asm
#AT&T x86 Assembly
     .file    "hello.c"
     .section .rodata
.LC0:
     .string  "Hello world!"
     .text
.globl main
     .type    main, @function
main:
     pushl    %ebp
     movl %esp, %ebp
     andl $-16, %esp
     subl $16, %esp
     movl $.LC0, (%esp)
     call puts
     movl $0, %eax
     leave
     ret
     .size    main, .-main
     .ident   "GCC: (GNU) 4.4.7 20120313 (Red Hat 4.4.7-4)"
     .section .note.GNU-stack,"",@progbits
```

Intel Syntax

- Microsoft (MASM), Intel, NASM
  - Type sizes are spelled out
    - DB – BYTE – 1 byte
    - DW – WORD – 2 bytes
    - DD – DWORD – 4 bytes (double word)
    - DQ – QWORD – 8 bytes (quad word)

  - First operand is destination
  - Second operand is source
  - Dereferences are denoted by [ ]

Intel Hello World

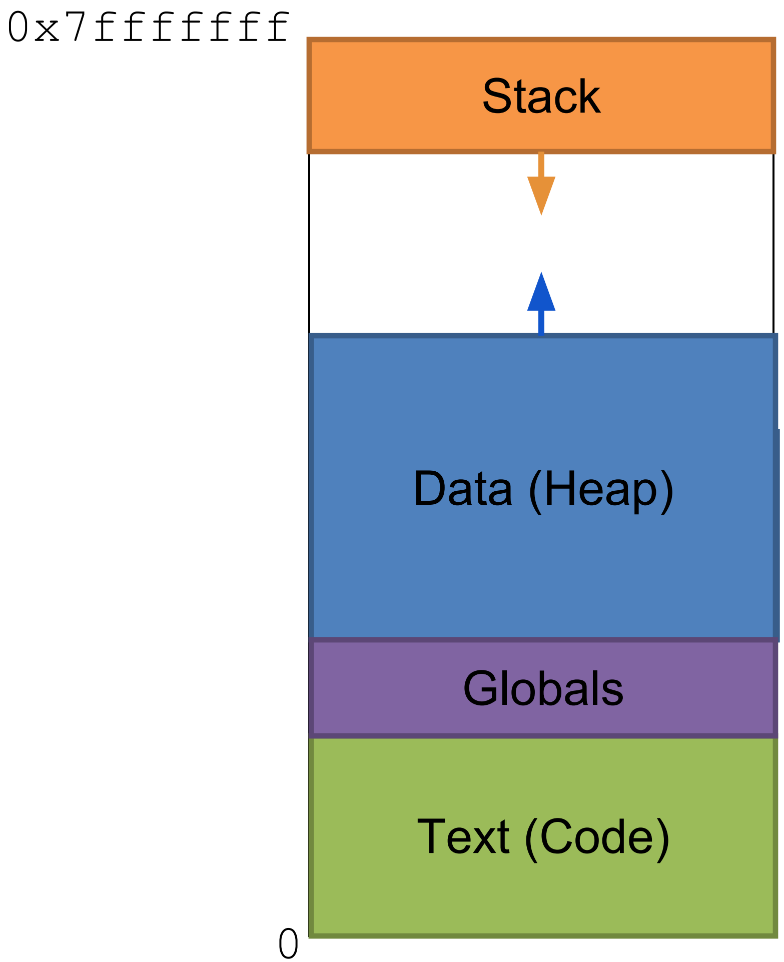
```c
//C program
int main()
{
	puts("Hello world!");
	return 0;
}
```

```
#Intel x86 Assembly
	.file	"hello.c"
	.intel_syntax noprefix
	.section .rodata
.LC0:
	.string	"Hello world!"
	.text
.globl main
	.type	main, @function
main:
	push	ebp
	mov	ebp, esp
	and	esp, -16
	sub	esp, 16
	mov	DWORD PTR [esp], OFFSET FLAT:.LC0
	call	puts
	mov	eax, 0
	leave
	ret
	.size	main, .-main
	.ident	"GCC: (GNU) 4.4.7 20120313 (Red Hat 4.4.7-4)"
	.section .note.GNU-stack,"",@progbits
```

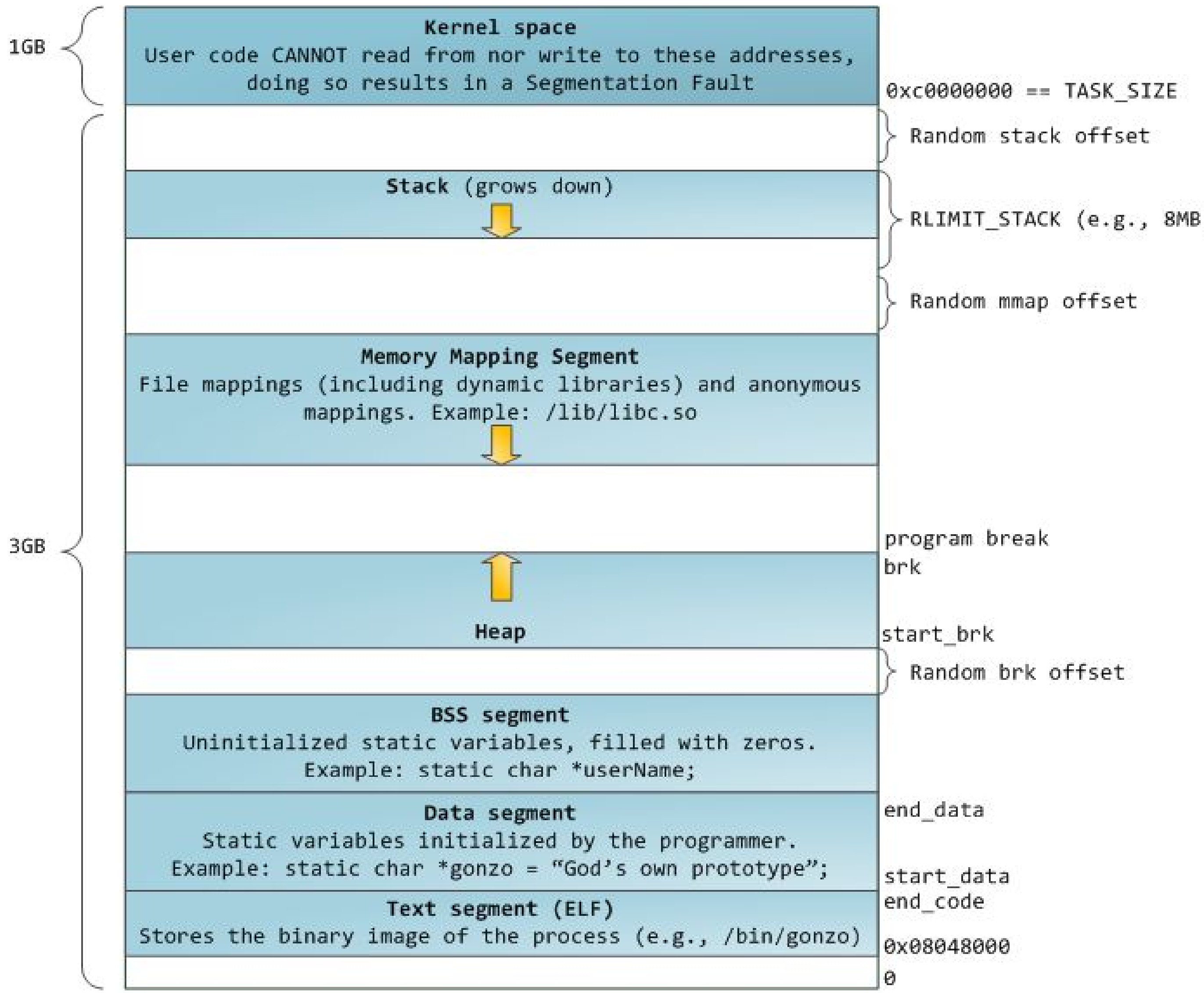
Stacks, Frames, and Calling Conventions

CS449 Spring 2016

Process's Address Space



Linux Address Space



Stack

- **Calling Convention**
  - An agreement, usually created by a system's designers, on how function calls should be implemented
- **Stack**
  - A portion of memory managed in a last-in, first-out (LIFO) fashion
- **Function Call**
  - A control transfer to a segment of code that ends with a return to the point in code immediately after where the call was made (the *return address*)

Activation Records

- An object containing all the necessary data for a function stored on the stack
  - Storage for Function parameters
  - Storage for Return address
  - Storage for Return value
  - Storage for Local variables
  - Storage for Temporaries (spilled registers)

- Also called a Stack Frame

Register Value Preservation

- Functions have dedicated stack storage but there is only one set of registers. How are they shared efficiently?

- Caller-Saved
  - A piece of data (e.g., a register) that must be explicitly saved if it needs to be preserved across a function call

- Callee-Saved
  - A piece of data (e.g., a register) that must be saved by a called function before it is modified, and restored to its original value before the function returns

MIPS Calling Convention

- First 4 arguments \$a0-\$a3
  - Remainder put on stack

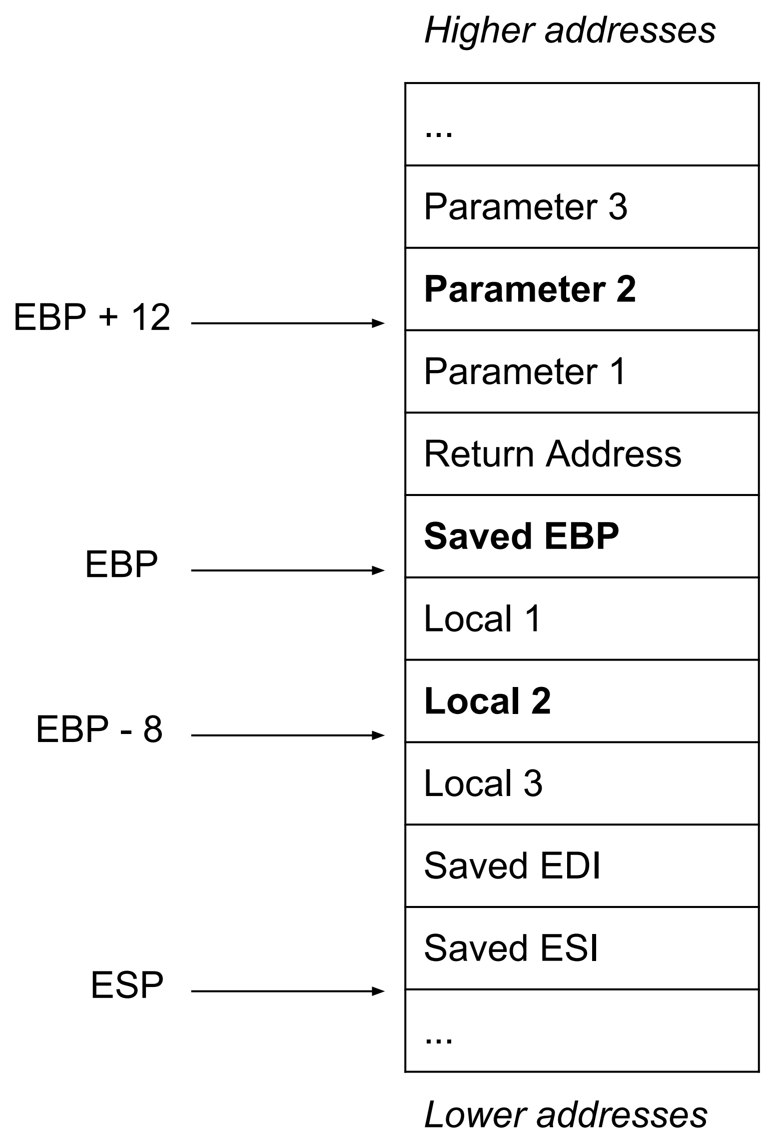
- Return values \$v0-\$v1

- \$t0-\$t9 are caller-saved temporaries
- \$s0-\$s9 are callee-saved

x86 Calling Convention

- **Arguments** (usually) passed on the stack
- **$EAX** is the return value
- **$EAX**, **$ECX**, and **$EDX** are generally caller-saved
- **$EBP**, **$EBX**, **$EDI**, and **$ESI** are generally callee-saved

x86 Stack


Higher addresses
...
Parameter 3
Parameter 2
Parameter 1
Return Address
Saved EBP
Local 1
Local 2
Local 3
Saved EDI
Saved ESI
...
Lower addresses
EBP + 12
EBP
EBP - 8
ESP

Remember this from Scoping?

```
#include <stdio.h>
int* foo() {
  int x = 5;
  return &x;
}
void bar() { int y = 10; }
int main()
{
  int *p = foo();
  printf("*p=%d\n", *p);
  bar();
  printf("*p=%d\n", *p);
  return 0;
}
```

```
>> gcc ./main.c
./main.c: In function ‘foo’:
./main.c:4: warning: function returns address of local variable
>> ./a.out
*p=5
*p=10
```

- The activation records for foo() and bar() landed on the same stack space