CS429: Computer Organization and Architecture

Instruction Set Architecture

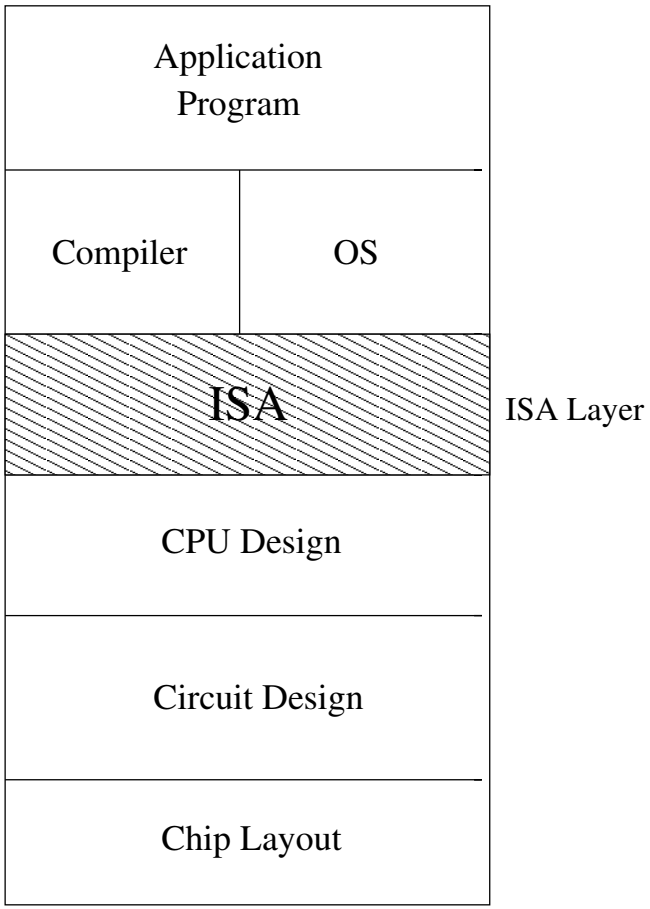
Warren Hunt, Jr. and Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: October 1, 2014 at 12:03

Topics of this Slideset

- Intro to Assembly language
- Programmer visible state
- Y86 Rudiments
- RISC vs. CISC architectures

Instruction Set Architecture



Assembly Language View

- Processor state: registers, memory, etc.
- Instructions and how instructions are encoded

Layer of Abstraction

- Above: how to program machine, processor executes instructions sequentially
- Below: What needs to be built
  - Use variety of tricks to make it run faster
  - E.g., execute multiple instuctions simultaneously

Why Y86?

The Y86 is a "toy" machine that is similar to the x86 but *much simpler*. It is a gentler introduction to assembly level programming than the x86.

- just a few instructions as opposed to thousands for the x86;
- fewer addressing modes;
- simpler system state;
- absolute addressing.

Everything you learn about the Y86 will apply to the x86 with very little modification.

Language / Machine Semantics

There are various means of giving a *semantics* or meaning to a programming system.

Probably the most sensible for an assembly (or machine) language is an *operational semantics*, also known as an *interpreter semantics*.

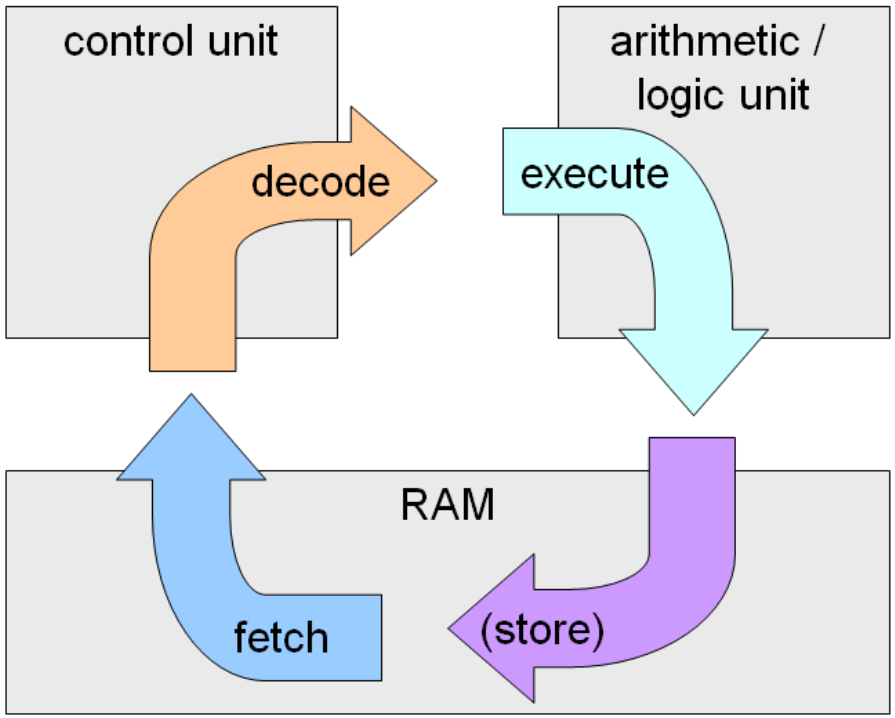
That is, we explain the semantics of each possible operation in the language by explaining the effect that execution of the operation has on the *machine state*.

Fetch / Decode / Execute Cycle

The most fundamental abstraction for the machine semantics for the x86/Y86 or similar machines is the *fetch-decode-execute* cycle.

The machine repeats the following steps forever:

1. fetch the next instruction from memory (the PC tells you which is next);
2. decode the instruction (in the control unit);
3. execute the instruction, updating the state appropriately;
4. go to step 1.



Conventions

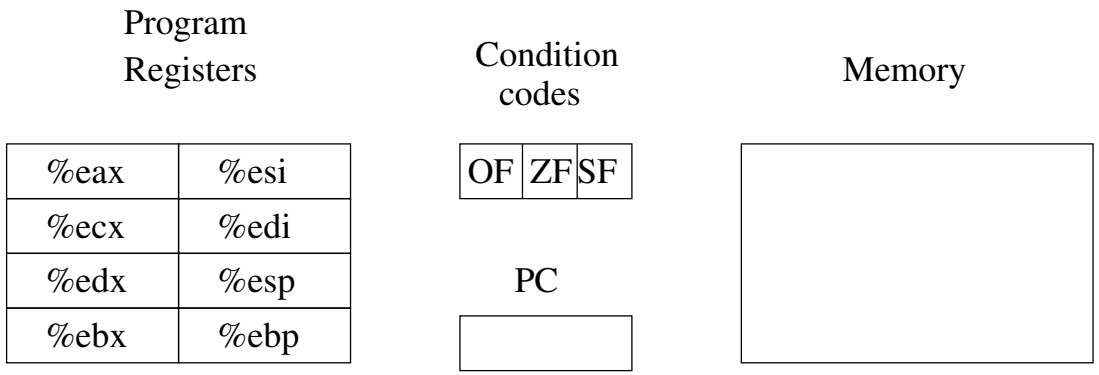
It's important to understand how individual operations update the system state. *But that's not enough!*

Much of the way the Y86/x86 operates is based on a a set of *programming conventions.* Without them, you won't understand how programs work, what the compiler generates, or how your code can interact with code written by others.

- How do you pass arguments to a procedure?
- Where are variables (local, global, static) created?
- How does a procedure return a value?
- How do procedures save and restore the state of the caller?

Some of these (e.g., the direction the stack grows) are reflected in specific machine operations; others are purely conventions.

Y86 Processor State



- Program registers: same 8 as IA32, each 32-bits
- Condition flags: 1-bit flags set by arithmetic and logical operations. OF: Overflow, ZF: Zero, SF: Negative
- Program counter: indicates address of instruction
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order
- Status code (not shown): status can be AOK, HLT, INS, ADR; indicate state of program execution.

Y86 Instructions

We're actually describing two languages: the assembly language and the machine language. There is (sort of) a 1-1 correspondence between them.

Format

- 1-6 bytes of information read from memory
  - Can determine instruction length from first byte
  - Not as many instruction types and simpler encoding than IA32
- Each instruction accesses and modifies some part(s) of the program state.

Encoding Registers

Each register has an associated 4-bit id:

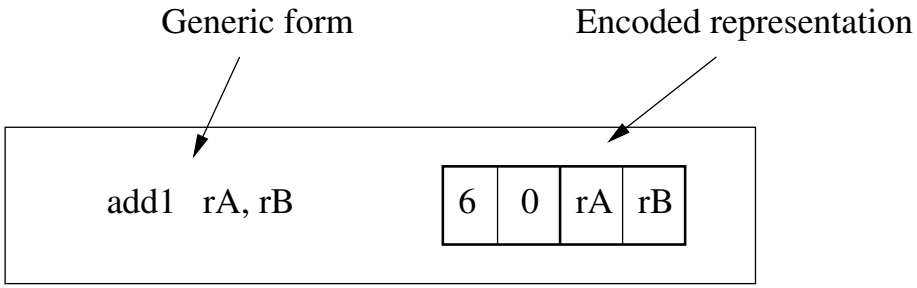
%eax	0	%esi	6
%ecx	1	%edi	7
%edx	2	%esp	4
%ebx	3	%ebp	5

Same encoding as in IA32. Register ID F indicates "no register." (Earlier versions of Y86 used 8 instead of F, so be on the lookout for places we forgot to switch. There are some in the book.)

Most of these registers are general purpose; %esp and %ebp have special functionality.

Instruction Example

Addition Instruction



- Add value in register rA to that in register rB.
  - Store result in register rB
  - Note that Y86 only allows addition to be applied to register data.
- E.g., `addl %eax, %esi` is encoded as: 60 06. Why?
- Set condition codes based on the result.
- Two byte encoding:
  - First indicates instruction type.
  - Second gives source and destination registers.

Arithmetic and Logical Operations

Add

`addl rA, rB`	6	0	rA	rB

Subtract (rA from rB)

`subl rA, rB`	6	1	rA	rB

And

`andl rA, rB`	6	2	rA	rB

Exclusive Or

`xorl rA, rB`	6	3	rA	rB

- Refer to generically as "OP1"
- Encodings differ only by "function code": lower-order 4-bits in first instruction byte.
- Set condition codes as side effect.

Move Operations

Register to Register

`rrmovl rA, rB`	2	0	rA	rB

Immediate to Register

`irmovl V, rB`	3	0	F	rB	V

Register to Memory

`rmmovl rA, D(rB)`	4	0	rA	rB	D

Memory to Register

`mrmovl D(rB), rA`	5	0	rA	rB	D

- Similar to the IA32 movl instruction.
- Similar format for memory addresses.
- Slightly different names to distinguish them.

Move Instruction Examples

IA32	Y86	Y86 Encoding
`movl $0xabcd, %edx`	`irmovl $0xabcd, %edx`	30 F2 cd ab 00 00
`movl %esp, %ebx`	`rrmovl %esp, %ebx`	20 43
`movl -12(%ebp), %ecx`	`mrmovl -12(%ebp), %ecx`	50 15 f4 ff ff ff
`movl %esi, 0x41c(%esp)`	`rmmovl %esi, 0x41c(%esp)`	40 64 1c 04 00 00
`movl %0xabcd, (%eax)`	none	
`movl %eax, 12(%eax, %edx)`	none	
`movl (%ebp, %ezx, 4), %ecx`	none	

The Y86 adds special move instructions to compensate for the lack of certain *addressing modes*.

Jump Instructions

Jump Unconditionally

`jmp Dest`	7	0	Dest

Jump when less or equal

`jle Dest`	7	1	Dest

Jump when less

`jl Dest`	7	2	Dest

Jump when equal

`je Dest`	7	3	Dest

Jump when not equal

`jne Dest`	7	4	Dest

Jump when greater or equal

`jge Dest`	7	5	Dest

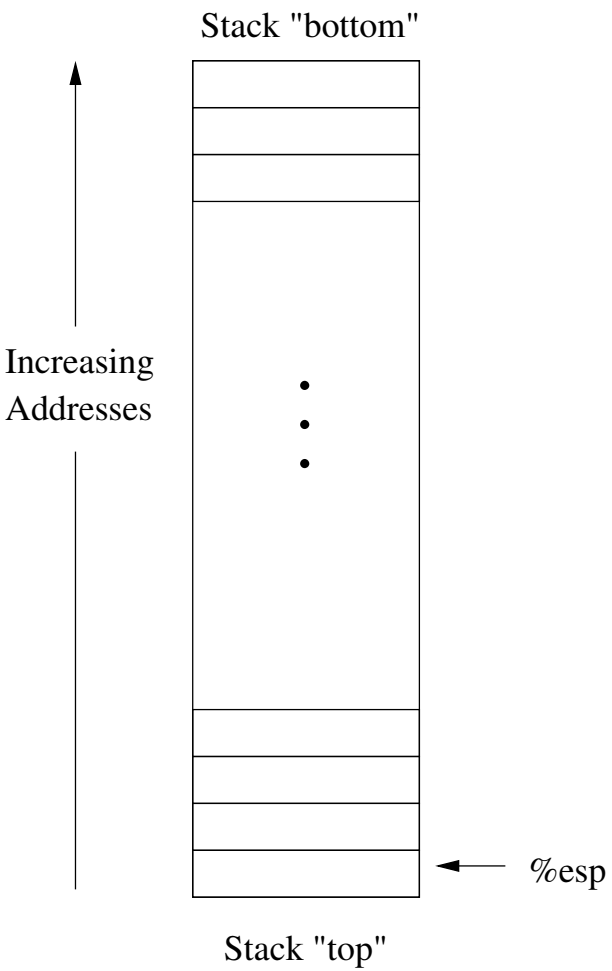
Jump when greater

`jg Dest`	7	6	Dest

Jump Instructions

- Refer to jump instructions generically as "jXX."
- Encodings differ only in "function code."
- Basically the same as the IA32 counterparts.
- Encode the full ("absolute") destination address; unlike the PC-relative addressing seen in IA32.

Y86 Program Stack



- Region of memory holding program data.
- Used in Y86 (and IA32) for supporting procedure calls.
- Stack top is indicated by %esp, address of top stack element.
- Stack grows toward lower addresses.
  - Top element is at lowest address in the stack.
  - When pushing, must first decrement stack pointer.
  - When popping, increment stack pointer.

Stack Operations

Push

pushl rA	a	0	rA	F

- Decrements %esp by 4.
- Store word from rA to memory at %esp.
- Similar to IA32 pushl operation.

Pop

popl rA	b	0	rA	F

- Read word from memory at %esp.
- Save in rA.
- Increment %esp by 4.
- Similar to IA32 popl operation.

Subroutine Call and Return

Subroutine call

`call Dest`	8	0	Dest

- Push address of next instruction onto stack.
- Start executing instructions at Dest.
- Similar to IA32 call instruction.

Subroutine return

`ret`	9	0

- Pop value from stack.
- Use as address for next instruction.
- Similar to IA32 ret instruction.

Miscellaneous Instructions

No operation

`nop`	1	0

- Don't do anything but advance PC.

Halt execution

`halt`	0	0

- Stop executing instructions.
- Sets status to HLT.
- IA32 has a comparable instruction, but you can't execute it in user mode.
- We will use it to stop the simulator.

Writing Y86 Code

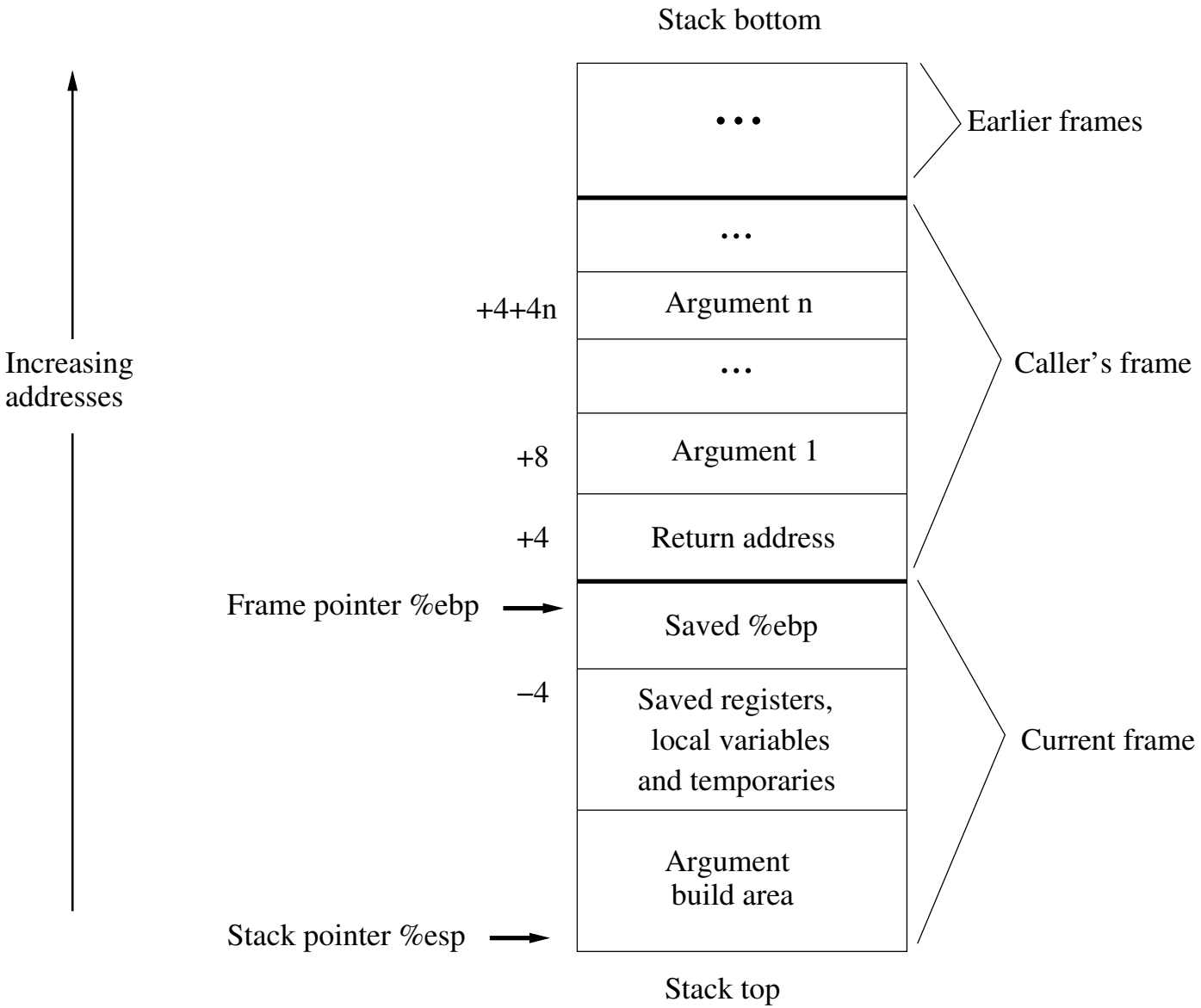
Try to use the C compiler as much as possible.

- Write code in C.
- Compile for IA32 with `gcc -S`.
- Transliterate into Y86 code.

To understand Y86 (or x86) code, you have to know the meaning of the statement, but also certain *programming conventions*, especially the *stack discipline*.

- How do you pass arguments to a procedure?
- Where are local variables created?
- How does a procedure return a value?
- How do procedures save and restore the state of the caller?

A Preview of the Stack



A Simple Example

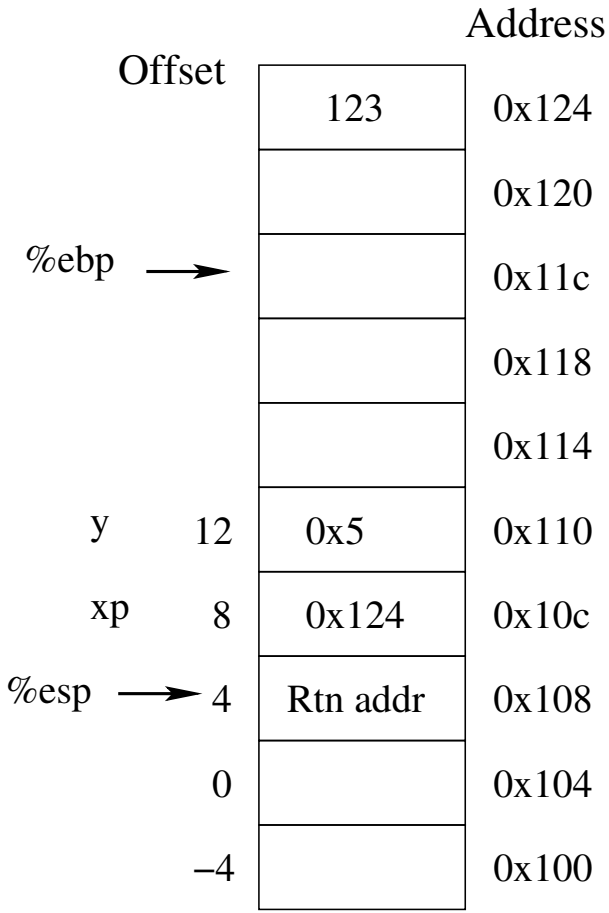
```
int simple( int *xp, int y )
{
    int t = *xp + y;
    *xp = t;
    return t;
}
```

Here's the corresponding Y86 code:

```
simple:
    pushl  %ebp                 # save frame pointer
    rrmovl %esp, %ebp           # create new frame pointer
    mrmovl 8(%ebp), %edx        # get xp
    mrmovl 0(%edx), %ebx        # get *xp
    mrmovl 12(%ebp), %eax       # get y
    addl   %ebx, %eax           # t = *xp + y
    rmmovl %eax, 0(%edx)        # store in *xp
    popl   %ebp                 # restore frame pointer
    ret                         # return to caller
```

Simple Example

```
simple:
    pushl   %ebp
    rrmovl  %esp, %ebp
    mrmovl  8(%ebp), %edx
    mrmovl  0(%edx), %ebx
    mrmovl  12(%ebp), %eax
    addl    %ebx, %eax
    rmmovl  %eax, 0(%edx)
    popl    %ebp
    ret
```



Another Example

```
/* Find number of elements in null-terminated list. */
int len2( int a[] )
{
    int len = 0;
    while (*a++)
        len++;
    return len;
}
```

How do you find the Y86 assembler that corresponds to this C code?

Could generate it by hand, or compile for x86 and translate.

Why not use arrays here instead of explicit pointers?

Arrays tend to use scaled addressing, which is available in x86 but not in Y86.

Another Example (2)

```
len2:   pushl   %ebp            # save frame pointer
        rrmovl  %esp, %ebp      # create new frame pointer
        mrmovl  8(%ebp), %edx   # get a
        mrmovl  0(%edx), %eax   # get *a
        xorl    %ecx, %ecx      # len = 0
        jmp     L26

L24:    mrmovl  0(%edx), %eax   # get *a
        irmovl  $1, %esi
        addl    %esi, %ecx      # len++

L26:    irmovl  $4, %esi
        addl    %esi, %edx      # a++
        andl    %eax, %eax      # *a == 0?
        jne     L24
        rrmovl  %ecx, %eax      # the return value
        rrmovl  %ebp, %esp      # clean frame
        popl    %ebp            # restore frame pointer
        ret                     # return to caller
```

Y86 Program Structure

```
        irmovl  Stack, %esp         # set up stack
        rrmovl %esp, %ebp           # set up frame
        irmovl  List, %edx
        pushl   %edx                # push arguments
        call    len2                # call function
        halt                        # stop execution
.align 4
List:                               # the array to
        .long   5043                # measure
        .long   6125
        .long   7395
        .long   0

# Function
len:    ....

# Allocate space for stack
.pos   0x100
Stack:
```

Y86 Program Structure (2)

- Program starts at address 0.
- Must set up the stack.
- Make sure that execution doesn't overwrite the code.
- Try to use symbolic names.
- Add assembler directives as appropriate.
- Caller pushes args onto the stack.
- Caller and/or callee must save registers that should be preserved.

Assembling a Y86 Program

```
unix> yas file.ys
```

- Generates object code `file.yo`.
- Actually looks like disassembler output.

```
0x000: 308400010000 |      irmovl  Stack, %esp
0x006: 2045         |      rrmovl  %esp, %ebp
0x008: 308218000000 |      irmovl  List, %edx
0x00e: a028         |      pushl   %edx
0x010: 8028000000   |      call    len2
0x015: 10           |      halt
0x018:              | .align  4
0x018:              | List:
0x018: b3130000     |      .long   5043
0x01c: ed170000     |      .long   6125
0x020: e31c0000     |      .long   7395
0x024: 00000000     |      .long   0
```

Simulating a Y86 Program

```
unix> yis file.yo
```

Instruction set simulator

- Computes the effect of each instruction on the processor state.
- Prints changes in state from original.

```
Stopped in 41 steps at PC = 0x16.  Exception 'HLT', CC
    Z=1 S=0 O=0
Changes to registers:
%eax:                           0x00000000 0x00000003
%ecx:                           0x00000000 0x00000003
%edx:                           0x00000000 0x00000028
%esp:                           0x00000000 0x000000fc
%ebp:                           0x00000000 0x00000100
%esi:                           0x00000000 0x00000004

Changes to memory:
0x00f4:                         0x00000000 0x00000100
0x00f8:                         0x00000000 0x00000015
0x00fc:                         0x00000000 0x00000018
```

Y86 Assembler

A program that translates Y86 code into machine language.

- 1-1 mapping of instructions to encodings.
- Resolves symbolic names.
- Translation is linear.
- Assembler directives give additional control.

Some common directives:

- `.pos x`: subsequent lines of code start at address x.
- `.align x`: align the next line to an x-byte boundary (e.g., long ints should be at a word address).
- `.long x`: put x at the current address; a way to initialize a value.

Y86 Stack Discipline

Stack used to:

- implement function calls;
- provide local storage;
- %esp (stack pointer) points to current top of stack.

ISA provides push and pop instructions.

- `push rA`: decrement %esp by 4; `MEM[%esp] = rA`
- `pop rA`: `rA = MEM[%esp]`; increment %esp by 4;

Each function has an associated frame.

- %ebp (frame pointer) points to its beginning.
- Holds return address, arguments, local variables.
- Provides storage for saved registers.

Y86 Stack Discipline (Continued)

Caller responsible for:

- Pushing arguments on stack in reverse order.
- Storing return address on stack.
- Transferring control to function.
- Cleaning up after function completes.

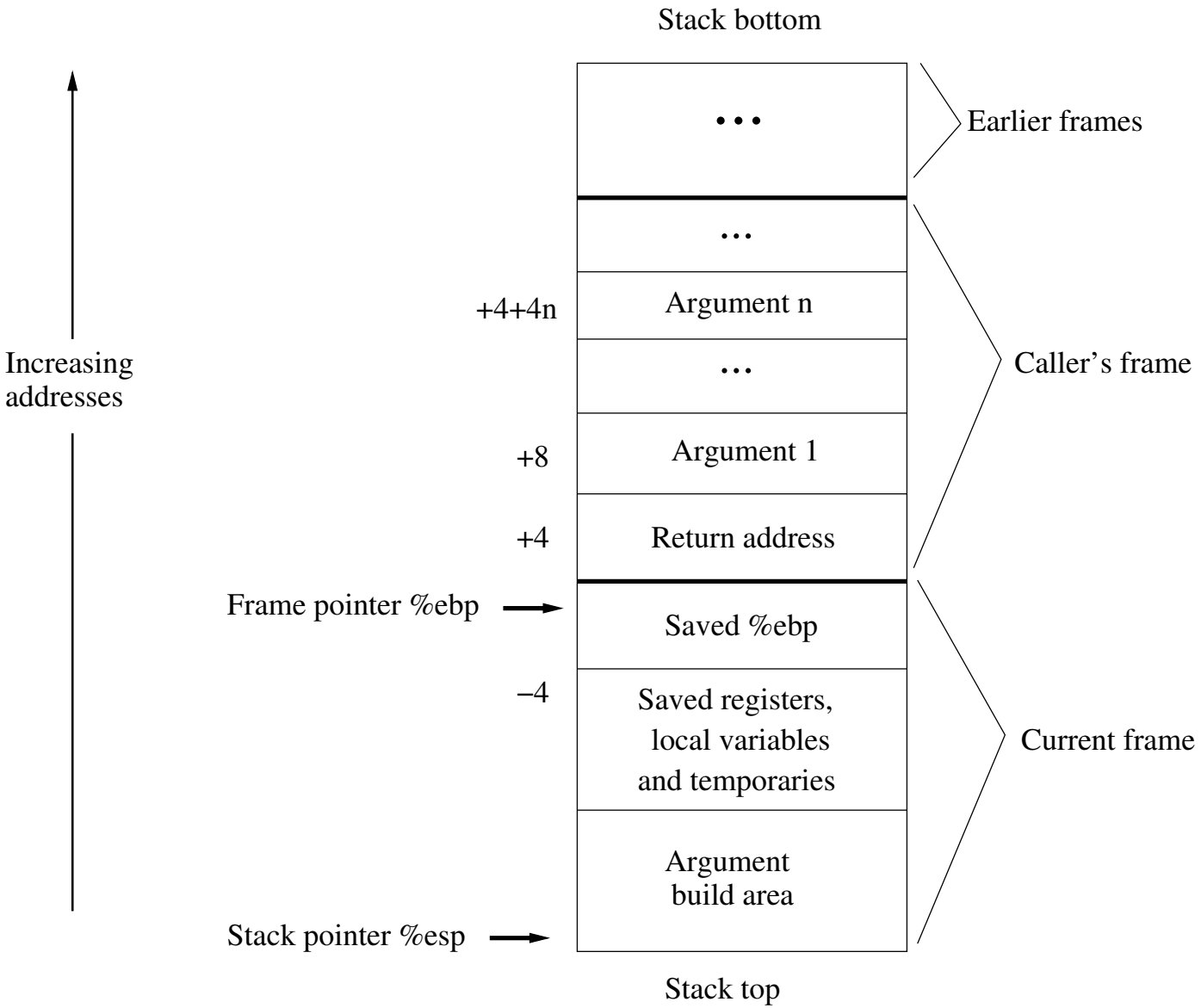
Callee responsible for:

- Saving %ebp.
- Setting %ebp to current top of stack.
- Saving registers it will be using.
- Upon completion of task, restoring %ebp and returning control to caller

ISA provides call and ret instructions.

- `call Dest`: push address of next instruction on stack; set PC to Dest.
- `ret`: pop stack; set PC to value popped.

Function Calls Using the Stack



A More Complete Example

```
int array[] = {0xd, 0xc0, 0xb00, 0xa000};

/* $begin sum-c */
int Sum( int *Start, int Count)
{
    int sum = 0;
    while (Count) {
        sum += *Start;
        Start++;
        Count--;
    }
    return sum;
}
/* $end sum-c */

int main()
{
    Sum( array, 4 );
    return 0;
}
```

The Compilation

```
# Execution begins at address 0
        .pos 0
init:
        irmovl  Stack, %esp         # set up stack pointer
        irmovl  Stack, %ebp         # set up frame pointer
        jmp     Main                # execute main program

# Array of 4 elements
array:
        .long 0xd
        .long 0xc0
        .long 0xb00
        .long 0xa000

Main:
        irmovl  $4, %eax
        pushl   %eax                # push 4
        irmovl  array, %edx
        pushl   %edx                # push array
        call    Sum                 # Sum( array, 4 )
        halt

# continues next slide
```

The Compilation (2)

```
#  int Sum( int *Start, int Count )
Sum:
        pushl   %ebp                # save old frame ptr.
        rrmovl  %esp, %ebp          # update frame ptr.
        mrmovl  8(%ebp), %ecx       # ecx = Start
        mrmovl  12(%ebp), %edx      # edx = Count
        irmovl  $0, %eax            # sum = 0
        andl    %edx, %edx
        je      End
Loop:
        mrmovl  0(%ecx), %esi       # get *Start
        addl    %esi, %eax          # add to Sum
        irmovl  $4, %ebx            #
        addl    %ebx, %ecx          # Start++
        irmovl  $-1, %ebx           #
        addl    %ebx, %edx          # Count -= 1
        jne     Loop
End:
        rrmovl  %ebp, %esp          # restore stack ptr.
        popl    %ebp                # restore frame ptr.
        ret

        .pos    0x100
Stack:                              # the stack goes here.
```

Complex Instruction Set Computer

- Dominant ISA style through the 80s.
- Lots of instructions:
  - Variable length
  - Stack as mechanism for supporting functions
  - Explicit push and pop instructions.
- ALU instructions can access memory.
  - E.g., `addl %eax, 12(%ebx)`
  - Requires memory read and write in one instruction execution.
  - Some ISAs had much more complex address calculations.
- Set condition codes as a side effect of other instructions.
- Basic philosophy:
  - Memory is expensive;
  - Instructions to support high-level language constructs.

RISC Instruction Sets

Reduced Instruction Set Computer

- Originated in IBM Research; popularized in Berkeley and Stanford projects.
- Few, simple instructions.
  - Takes more instructions to execute a task, but faster and simpler implementation
  - Fixed length instructions for simpler decoding
- Register-oriented ISA
  - More registers (32 typically)
  - Stack is back-up for registers
- Only load and store instructions can access memory (mrmovl and rmmovl in Y86).
- Explicit test instructions set condition codes.
- Philosophy: KISS

MIPS Instruction Format

Register–register:

Op	Ra	Rb	Rd	00000	Fn

```
addu $3,$2,$1      # register add: $3 = $2+$1
```

Register–immediate:

Op	Ra	Rb	Offset

```
addu $3,$2,3145      # immediate add: $3 = $2+3145
sll  $3,$2,2         # shift left: $3 = $2 << 2
```

Branch:

Op	Ra	Rb	Immediate

```
beq  $3,$2,dest      # Branch when $3 = $2
```

Load/Store:

Op	Ra	Rb	Immediate

```
lw   $3,16($2)     # Load word: $3 = M[$2+16]
sw   $3,16($2)     # Store word: M[$2+16] = $3
```

MIPS Registers

bf Name	**Number**	**Use**	**Callee preserves?**
$zero	$0	constant 0	N/A
$at	$1	assembler temporary	No
$v0–$v1	$2–$3	function returns expression evaluation	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	N/A
$gp	$28	global pointer	Yes
$sp	$29	stack pointer	Yes
$fp	$30	frame pointer	Yes
$ra	$31	return address	N/A

What To Do?

- In the 1980s a nasty debate:
  - Direct compilation vs. optimized compilation
  - Support for hardware management vs. simpler control
- Several startups (ARM, MIPS) with technically superior products
- Decisions based on non-technical factors
  - Money makes the world go round
  - Need for backward compatibility
- Enough transistors on a chip to achieve high performance
- Intel seems to be moving away from x86 legacy.