# Machine-level representation of programs: control, comparisons, branching, loops

Yipeng Huang

Rutgers University

March 24, 2022

# Table of contents

# Announcements

## PA4 bomb lab

- PA4 bomb lab out and live. Due Tuesday, April 5.

## Short quiz next week

Short quiz on assembly basics and control spanning Tuesday 3/29 to Thursday 3/31.

## Class session plan

- Today, Thursday, 3/24: Control flow (conditionals, if, for, while, do loops) in assembly. (Book chapter 3.6)
- Tuesday, 3/29: Function calls in assembly. (Book chapter 3.7)
- Thursday, 3/31: Arrays and data structures in assembly. (Book chapter 3.8)

# Table of contents

# What is control flow?

## Control flow is:

- Change in the sequential execution of instructions.
- Change in the steady incrementation of the program counter / instruction pointer (%rip register).

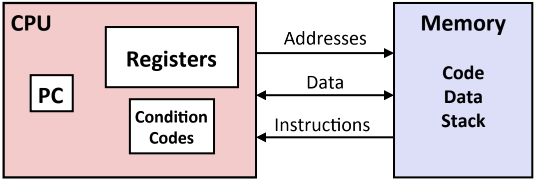
## Control primitives in assembly build up to enable C and Java control statements:

- if-else statements
- do-while loops
- while loops
- for loops
- switch statements

# Condition codes



## Assembly/Machine Code View




### Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

# Condition codes

Automatically set by most arithmetic instructions.

| Applicable types | Condition code | Name | Use |
|---|---|---|---|
| Signed and unsigned | `ZF` | Zero flag | The most recent operation yielded zero. |
| Unsigned types | `CF` | Carry flag | The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations |
| Signed types | `SF` | Sign flag | The most recent operation yielded a negative value. |
| Signed types | `OF` | Overflow flag | The most recent operation yielded a two's complement positive or negative overflow. |

Table: Condition codes important for control flow

# Comparison instructions

`cmpq source1, source2`

Performs source2 – source1, and sets the condition codes without setting any destination register.

# Set instructions

`cmp source1, source2` performs source2 – source1, sets condition codes.

| Applicable types | Set instruction | Logical condition | Intutive condition |
|---|---|---|---|
| Signed and unsigned | `sete` / `setz` | `ZF` | Equal / zero |
| Signed and unsigned | `setne` / `setnz` | ~ `ZF` | Not equal / not zero |
| Unsigned | `setb` / `setnae` | `CF` | Below |
| Unsigned | `setbe` / `setna` | `CF`\|`ZF` | Below or equal |
| Unsigned | `seta` / `setnbe` | ~ `CF`& ~ `ZF` | Above |
| Unsigned | `setnb` / `setae` | ~ `CF` | Above or equal |
| Signed | `sets` | `SF` | Negative |
| Signed | `setns` | ~ `SF` | Nonegative |
| Signed | `setl` / `setnge` | `SF ^ OF` | Less than |
| Signed | `setle` / `setng` | (`SF ^ OF`)\|`ZF` | Less than or equal |
| Signed | `setg` / `setnle` | ~ (`SF ^ OF`)& ~ `ZF` | Greater than |
| Signed | `setge` / `setnl` | ~ (`SF ^ OF`) | Greater than or equal |

Table: Set instructions

# Side review: De Morgan's laws

- $\neg A \wedge \neg B \iff \neg(A \vee B)$
- $(\sim A)\&(\sim B) \iff \sim (A|B)$

# Test for equality

```c
short equal_sl (
    long x,
    long y
) {
    return x==y;
}
```

C code function above translates to the assembly on the right.

```
equal_sl:
    xorl %eax, %eax
    cmpq %rsi, %rdi
    sete %al
    ret
```

## Explanation

- `xorl %eax, %eax`: Zeros the 32-bit register %eax.
- `cmpq %rsi, %rdi`: Calculates %rdi − %rsi ($x - y$), sets condition codes without updating any destination register.
- `sete %al`: Sets the 8-bit %al subset of %eax if op yielded zero.

# Test if unsigned x is below unsigned y

```c
short below_ul (
    unsigned long x,
    unsigned long y
) {
    return x<y;
}
```

```c
short nae_ul (
    unsigned long x,
    unsigned long y
) {
    return !(x>=y);
}
```

Both C code functions above translate to the assembly on the right.

```
below_ul:
nae_ul:
    xorl %eax, %eax
    cmpq %rsi, %rdi
    setb %al
    ret
```

## Explanation

- `xorl %eax, %eax`: Zeros %eax.
- `cmpq %rsi, %rdi`: Calculates %rdi − %rsi ($x - y$), sets condition codes without updating any destination register.
- `setb %al`: Sets %al if CF flag set indicating unsigned overflow.

# Table of contents

# Jump instructions



## Jumping

- **jX Instructions**
  - Jump to different part of code depending on condition codes

| jX | Condition | Description |
|---|---|---|
| jmp | 1 | Unconditional |
| je | ZF | Equal / Zero |
| jne | ~ZF | Not Equal / Not Zero |
| js | SF | Negative |
| jns | ~SF | Nonnegative |
| jg | ~(SF^OF)&~ZF | Greater (Signed) |
| jge | ~(SF^OF) | Greater or Equal (Signed) |
| jl | (SF^OF) | Less (Signed) |
| jle | (SF^OF)\|ZF | Less or Equal (Signed) |
| ja | ~CF&~ZF | Above (unsigned) |
| jb | CF | Below (unsigned) |

# Branch statements

```
unsigned long absdiff_ternary (
    unsigned long x, unsigned long y ){
        return x<y ? y-x : x-y;
}
```

```
unsigned long absdiff_if_else (
    unsigned long x, unsigned long y ){
        if (x<y) return y-x;
        else return x-y;
}
```

```
unsigned long absdiff_goto (
    unsigned long x, unsigned long y ){
        if (!(x<y)) goto Else;
        return y-x;
    Else:
        return x-y;
}
```

All C functions above translate (-fno-if-conversion) to assembly at right.

```
absdiff_if_else:
absdiff_goto:
    cmpq %rsi, %rdi
    jnb .ELSE
    movq %rsi, %rax
    subq %rdi, %rax
    ret
.ELSE:
    movq %rdi, %rax
    subq %rsi, %rax
    ret
```

## Explanation

- `cmpq %rsi, %rdi`: Calculates %rdi − %rsi ($x - y$), sets condition codes.
- `jnb .ELSE`: Sets program counter / instruction pointer in %rip (.ELSE) if CF flag not set indicating no unsigned overflow.

# Table of contents

# Using GDB to carefully step through execution of the bomb program

`gdb bomb`

## Setting breakpoints and running / stepping through code

- `break explode_bomb` or `b explode_bomb`: Pause execution upon entering explode_bomb function.
- `break phase_1` or `b phase_1`: Pause execution upon entering phase_1 function.
- `run mysolution.txt` or `r mysolution.txt`: Run the code passing the solution file.
- `continue` or `c`: Continue until the next breakpoint.
- `nexti` or `ni`: Step one instruction, but proceed through subroutine calls.
- `stepi` or `si`: Step one instruction exactly. Steps into functions / subroutine calls.

# Example phase_1 in example bomb from CS:APP website

```
0000000000400ee0 <phase_1>:
  400ee0: 48 83 ec 08          sub    $0x8,%rsp
  400ee4: be 00 24 40 00       mov    $0x402400,%esi
  400ee9: e8 4a 04 00 00       callq  401338 <strings_not_equal>
  400eee: 85 c0                test   %eax,%eax
  400ef0: 74 05                je     400ef7 <phase_1+0x17>
  400ef2: e8 43 05 00 00       callq  40143a <explode_bomb>
  400ef7: 48 83 c4 08          add    $0x8,%rsp
  400efb: c3                   retq
```

## Understanding what we're seeing here

- Don't let `callq` to `explode_bomb` at instruction address `400ef2` happen...
- so, must ensure `je` instruction does jump, so we want `test` instruction to set ZF condition code to 0.
- so, must ensure `callq` to `strings_not_equal()` function returns 0.

# Using GDB to carefully step through execution of the bomb program

`gdb bomb`

## Printing and examining registers and memory addresses

- `print /x $eax` or `p /x $eax`: Print value of %eax register as hex.
- `print /d $eax` or `p /d $eax`: Print value of %eax register as decimal.
- `x /s 0x402400`: Examine memory address 0x402400 as a string.

# Table of contents

# Deep CPU pipelines

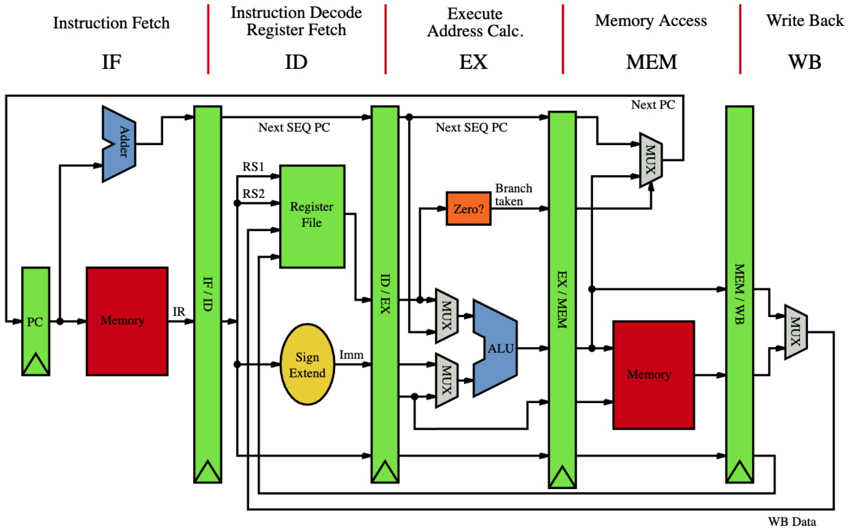



Figure: Pipelined CPU stages. Image credit wikimedia

# Table of contents

# Compiling for loops to while loops

C loop statements such as for loops, while loops, and do-while loops do not exist in assembly. They are instead constructed from conditional jump statements.

```
unsigned long count_bits_for (
  unsigned long number
) {
  unsigned long tally = 0;
  for (
    int shift=0; // init
    shift<8*sizeof(unsigned long); // test
    shift++ // update
  ) {
    // body
    tally += 0b1 & number>>shift;
  }
  return tally;
}
```

```
unsigned long count_bits_while (
  unsigned long number
) {
  unsigned long tally = 0;
  int shift=0; // init
  while (
    shift<8*sizeof(unsigned long) // test
  ) {
    // body
    tally += 0b1 & number>>shift;
    shift++; // update
  }
  return tally;
}
```

# Compiling while loops to do-while loops

```c
unsigned long count_bits_while (
  unsigned long number
) {
  unsigned long tally = 0;
  int shift=0; // init
  while (
    shift<8*sizeof(unsigned long) // test
  ) {
    // body
    tally += 0b1 & number>>shift;
    shift++; // update
  }
  return tally;
}
```

```c
unsigned long count_bits_do_while (
  unsigned long number
) {
  unsigned long tally = 0;
  int shift=0; // init
  do {
    // body
    tally += 0b1 & number>>shift;
    shift++; // update
  } while (shift<8*sizeof(unsigned long)); // test
  return tally;
}
```

If initial iteration is guaranteed to run, then do one fewer test.

# Compiling do-while loops to goto statements

```c
unsigned long count_bits_do_while (
  unsigned long number
) {
  unsigned long tally = 0;
  int shift=0; // init
  do {
    // body
    tally += 0b1 & number>>shift;
    shift++; // update
  } while (shift<8*sizeof(unsigned long)); // test
  return tally;
}
```

```c
unsigned long count_bits_goto (
  unsigned long number
) {
  unsigned long tally = 0;
  int shift=0; // init
LOOP:
  // body
  tally += 0b1 & number>>shift;
  shift++; // update
  if (shift<8*sizeof(unsigned long)) { // test
    goto LOOP;
  }
  return tally;
}
```

Loops get compiled into goto statements which are readily translated to assembly.

# Compiling goto statements to assembly conditional jump instructions

```
unsigned long count_bits_goto (
  unsigned long number
) {
  unsigned long tally = 0;
  int shift=0; // init
LOOP:
  // body
  tally += 0b1 & number>>shift;
  shift++; // update
  if (shift<8*sizeof(unsigned long)) { // test
    goto LOOP;
  }
  return tally;
}
```

All C loop statements so far translate to assembly at right.

```
count_bits_for:
count_bits_while:
count_bits_do_while:
count_bits_goto:
  xorl %ecx, %ecx # int shift=0; // init
  xorl %eax, %eax # unsigned long tally = 0;
.LOOP:
  movq %rdi, %rdx # number
  shrq %cl, %rdx  # number>>shift
  incl %ecx       # shift++; // update
  andl $1, %edx.  # 0b1 & number>>shift
  addq %rdx, %rax # tally += 0b1 & number>>shi
  cmpl $64, %ecx  # shift<8*sizeof(unsigned lo
  jne .LOOP       # goto LOOP;
  ret             # return tally;
```

# Table of contents