Machine Language

CS 3330

Samira Khan
University of Virginia
Feb 2, 2017

AGENDA

- Logistics
- Review of Abstractions
- Machine Language

Logistics

- Feedback
  - Not clear
  - Hard to hear
  - Use microphone
  - Good feedback

- I'm just confused how you think that you've prepared us for the homework or lab. In class today we learned about logical ANDs. Why the fuck are we learning that? Teach us assembly or something else that is actually useful.
  - I agree these are easy; May be can get rid of binary operations next time
  - Reviewing is still useful
  - More useful things to come throughout the semester
  - The goal is to learn

Logistics

- Declaring BACS major
- Applications are open to apply to declare the BACS
- Deadline is on Feb 20, 2017
- Full details on how to apply are found here:
- http://bit.ly/apply-bacs-s17

Logistics

- Next Lab: In-lab quiz
  - on lab machines
  - during assigned lab times
  - without assistance (TAs will be proctoring)
- Coding assignment: strlen, strsep
- https://archimedes.cs.virginia.edu/cs3330/c/lab1.php
  - Interface is available outside of lab
  - But only work done in lab will count
  - We check IPs + submission times
  - ”It's okay if you memorize the code, but we don't think that's best strategy"
- No homework due before it -- the homework is prepping for it
- Follow-up "lists in C" homework assignment

Reading for Next Week

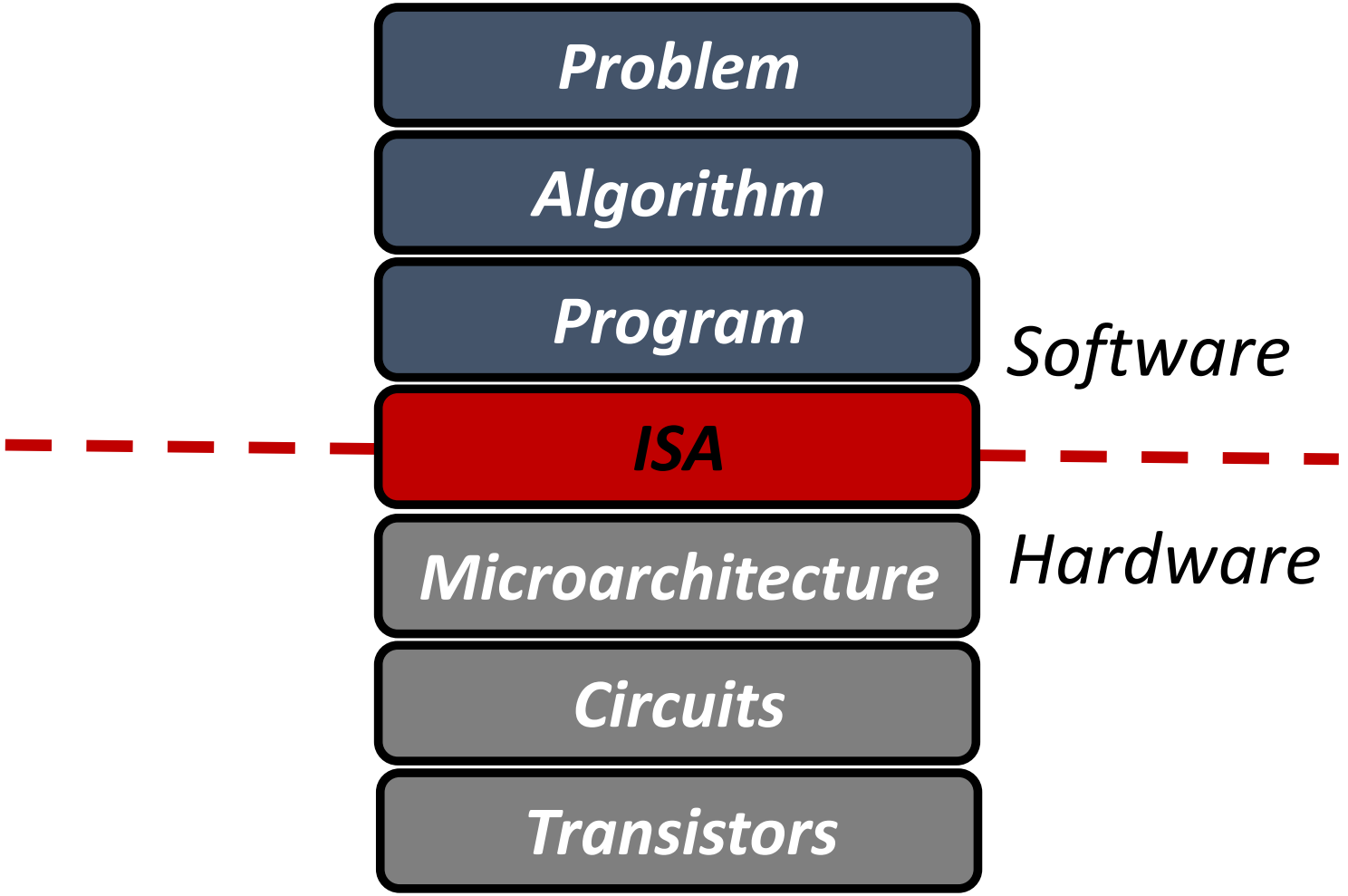
- **ISA's and Y86-64**
- §4.1

- **Sequential Processor**
- §4.2; §4.3.1

AGENDA

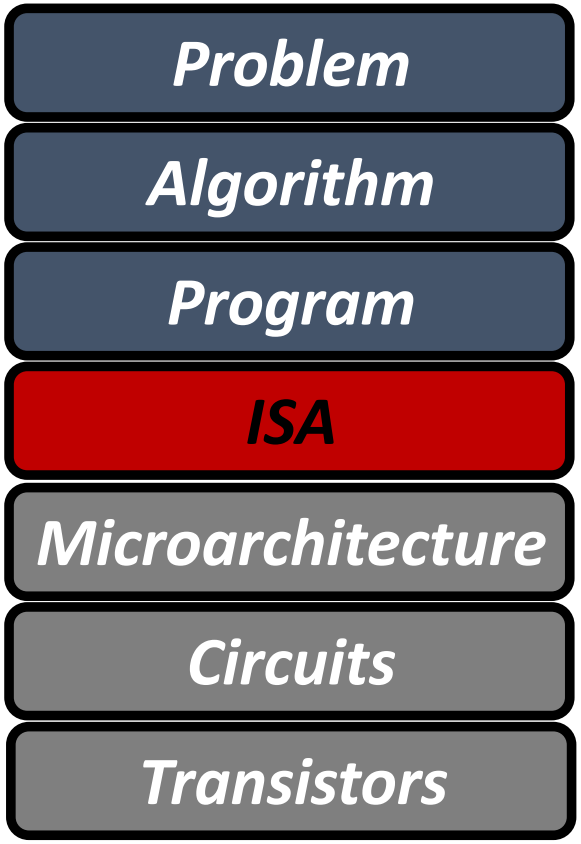
- Logistics
- Review of Abstractions
- Machine Language

LEVELS OF TRANSFORMATION



What is Computer Architecture

- ISA (Instruction Set Architecture)
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write and debug system/user programs
- Microarchitecture
  - Specific implementation of an ISA
  - Not visible to the software
- Microprocessor
  - **ISA, uarch**, circuits
  - "Architecture" = ISA + microarchitecture



How to interpret zeros and ones?

- At the lowest level, all the machine sees are zeros and ones

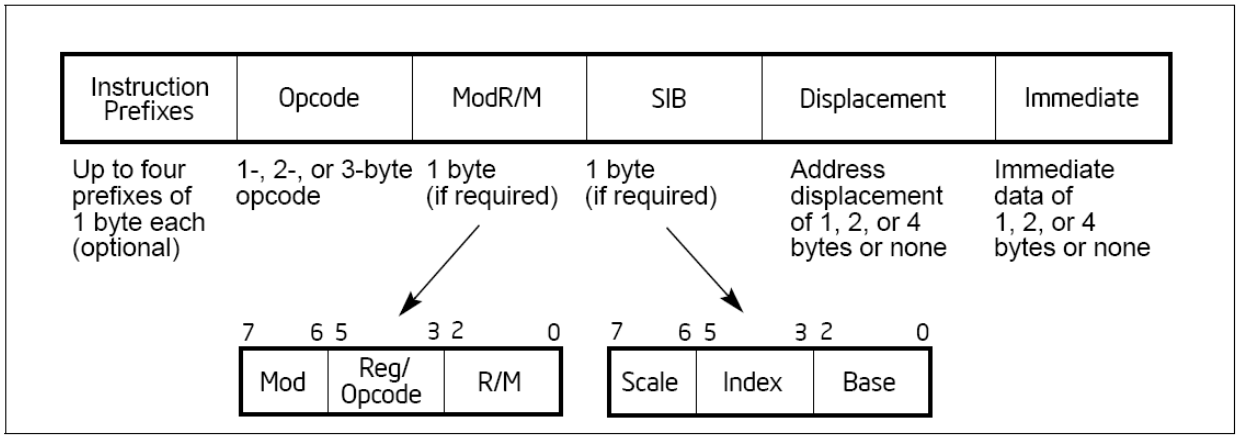
- How to interpret this strings of zeros and ones?
- ISA determines how to interpret the software instructions
  - So that hardware can understand those
  - Each bit in the instruction means different things

Intel® 64 and IA-32 Architectures
Software Developer's Manual

Volume 1:
Basic Architecture

0110 0000 0000 0010 → What does it mean?

x86 format


Instruction Prefixes
Opcode
ModR/M
SIB
Displacement
Immediate
Up to four prefixes of 1 byte each (optional)
1-, 2-, or 3-byte opcode
1 byte (if required)
1 byte (if required)
Address displacement of 1, 2, or 4 bytes or none
Immediate data of 1, 2, or 4 bytes or none
7 6 5 3 2 0
Mod
Reg/ Opcode
R/M
7 6 5 3 2 0
Scale
Index
Base

ARM

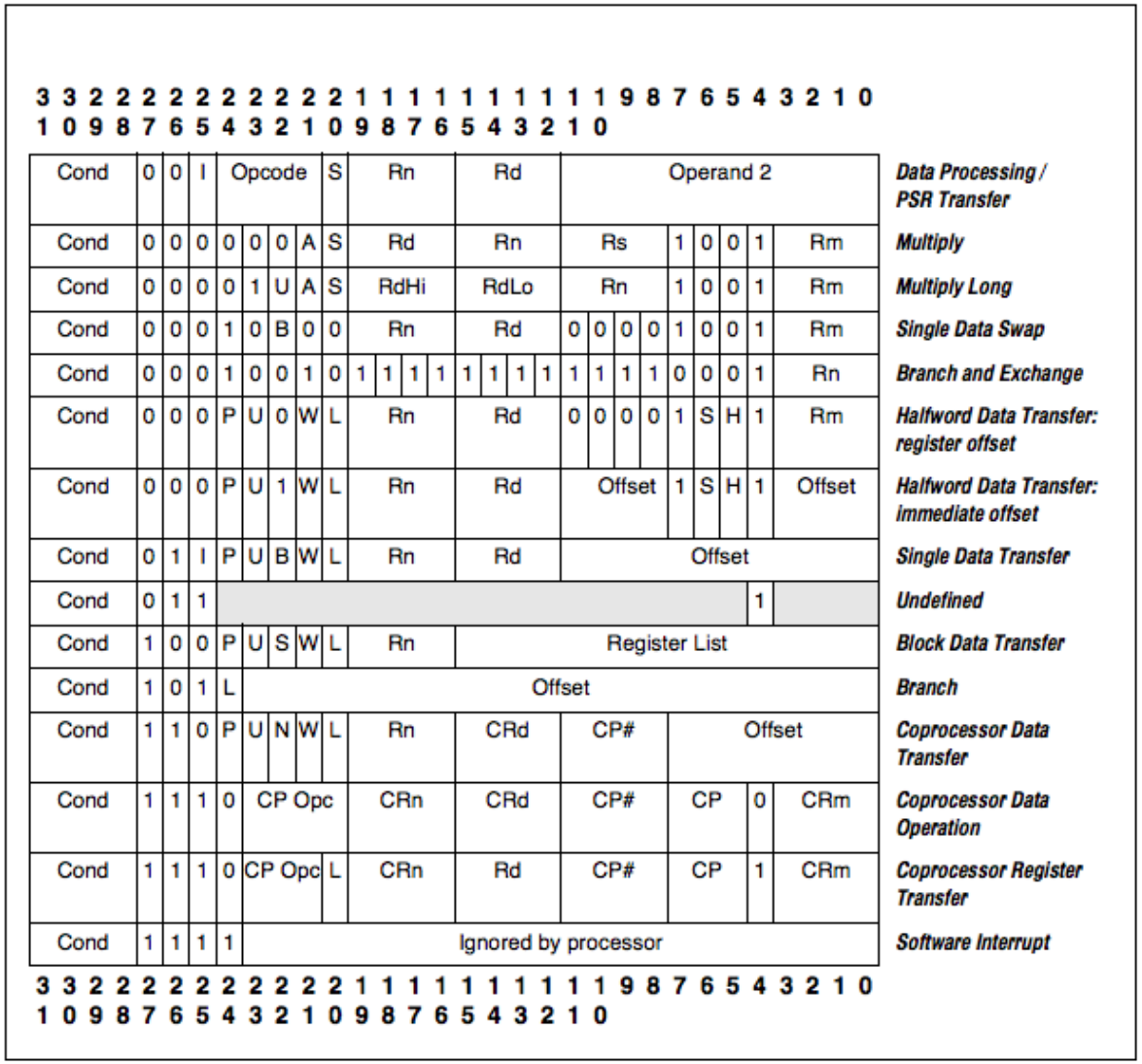
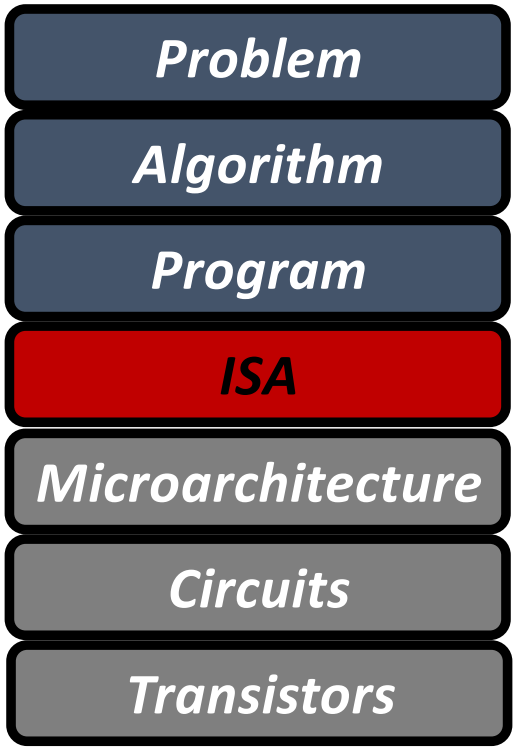


Figure 4-1: ARM instruction set formats

So what language we can use that resembles the machine language?

- 0110 0000 0000 0010 → What does it mean?

- x += y → "Higher-level" language: C
- add %rbx, %rax → Assembly: X86-64
- 60 03 → Machine code



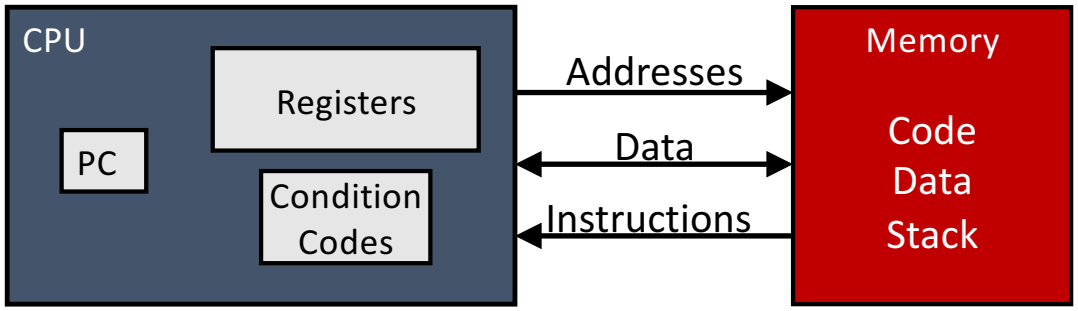
Summary

- Microarchitecture
  - Specific implementation of an ISA
  - Not visible to the software
- Microprocessor
  - **ISA, uarch**, circuits
  - "Architecture" = ISA + microarchitecture
- Example ISAs:
  - Intel: x86, x86-64
  - ARM: Used in almost all mobile phones
- Code Forms:
  - Machine Code: The byte-level programs that a processor executes
  - Assembly Code: A text representation of machine code

AGENDA

- Logistics
- Review of Abstractions
- Machine Language

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

Why do we need PC?

The Von Neumann Model/Architecture

- Also called *stored program computer* (instructions in memory). Two key properties:

- Stored program
  - Instructions stored in a linear memory array
  - Memory is unified between instructions and data
    - The interpretation of a stored value depends on the control signals

- Sequential instruction processing
  - One instruction processed (fetched, executed, and completed) at a time
  - Program counter (instruction pointer) identifies the current instr.
  - Program counter is advanced sequentially except for control transfer instructions

A Sample Program

Address of the function sumstore

```
0000000000400595 <sumstore>:
  400595:  53                   push   %rbx
  400596:  48 89 d3             mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff       callq  400590 <plus>
  40059e:  48 89 03             mov    %rax,(%rbx)
  4005a1:  5b                   pop    %rbx
  4005a2:  c3                   retq
```

PC will store the address of the next instruction

Assembly Language

- Operations
- Data types
- Registers
- Addressing modes
- Branches

Assembly Language: Operations

- Arithmetic
  - Perform arithmetic function on register or memory data
- Data Transfer
  - Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Control
  - Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Instructions

- **`movq` *Source*, *Dest*:**
- movq $42, (%rbx)
  - memory[rbx] ← 42
- subq %rax, %rbx
  - rbx ← rbx – rax
- addq 0x1000, %rax
  - rax ← rax + memory[0x1000]
- addq $0x1000, %rax
  - rax ← rax + 0x1000

Assembly Language

- Operations
- Data types
- Registers
- Addressing modes
- Branches

Data Types

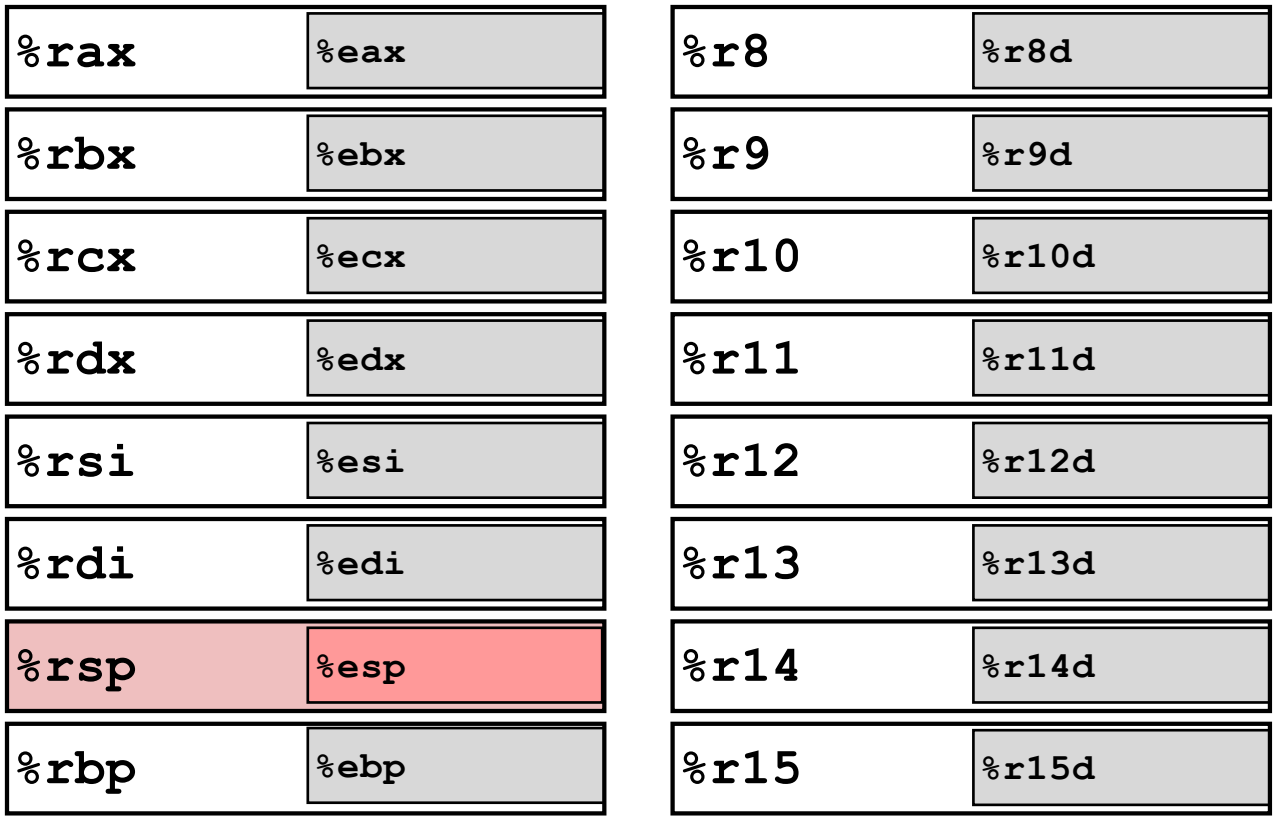
- movq $42, (%rbx)
  - memory[rbx] ← 42
- q indicates length (8 bytes)
- l: 4; w: 2; b: 1
- movl, movw, movb

Type Specifier	Bytes addressed
BYTE	1
WORD	2
DWORD	4
QWORD	8

Assembly Language

- Operations
- Data types
- Registers
- Addressing modes
- Branches

x86-64 Integer Registers



Assembly Language

- Operations
- Data types
- Registers
- Addressing modes
- Branches

Addressing Modes

- Most General Form

  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]

  - D: Constant “displacement” 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for **`%rsp`**
  - S: Scale: 1, 2, 4, or 8

- Special Cases

  (Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]

  D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D]

  (Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`		
`(%rdx,%rcx)`		
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`		
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`		

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Why so many modes?

- Advantage of more addressing modes:
  - Enables better mapping of high-level constructs to the machine: some accesses are better expressed with a different mode → reduced number of instructions and code size
    - Think array accesses
- Disadvantage:
  - More work for the compiler
  - More work for the microarchitect
- (Remember that these addressing modes need to be supported by the ISAs and implemented in the microarchitecture)

Assembly Language

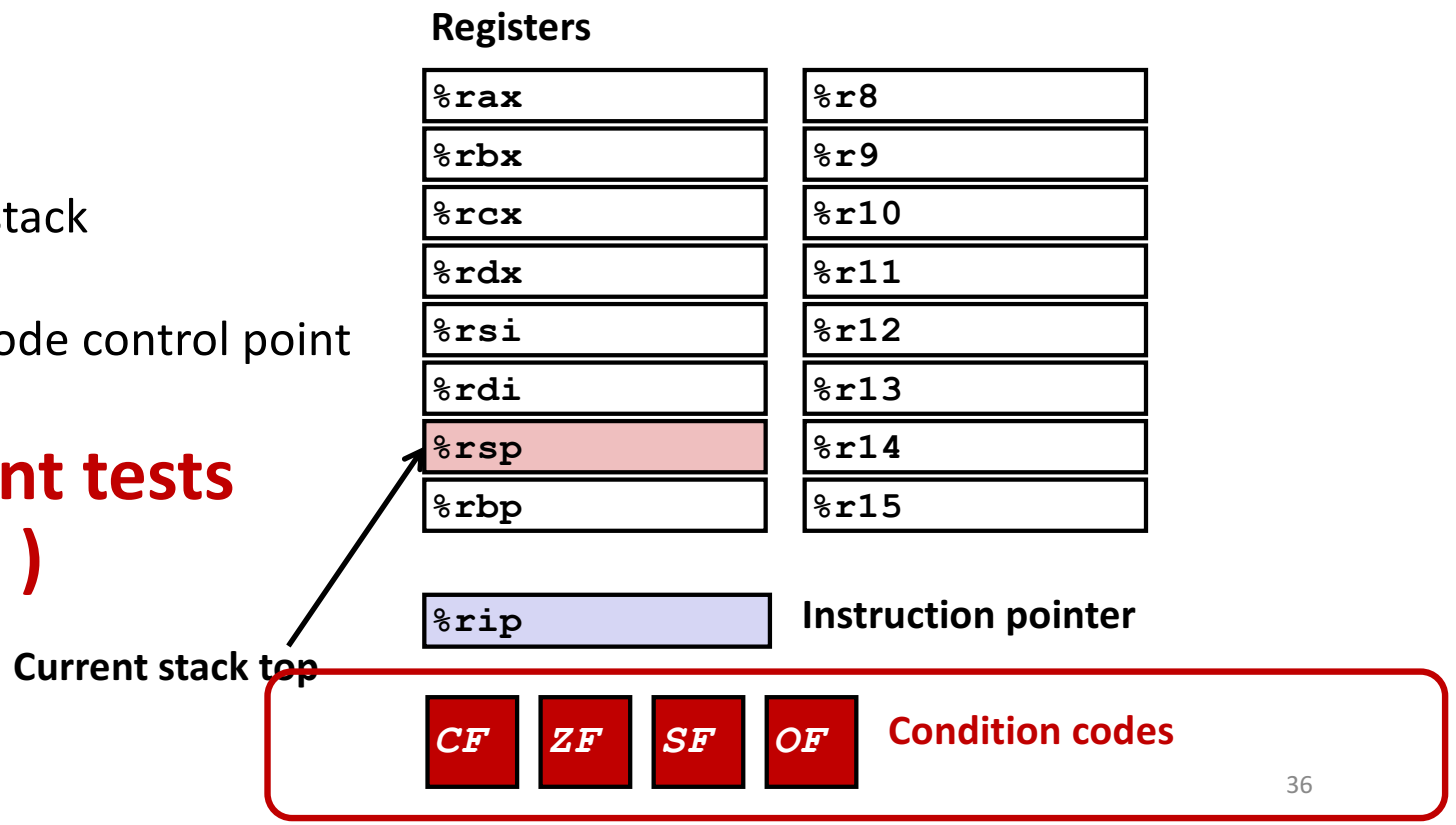
- Operations
- Data types
- Registers
- Addressing modes
- Branches

Branches

- Control: Condition codes
- Conditional branches
- Loops

Review: Processor State (x86-64)

- Information about currently executing program
  - Temporary data (**`%rax`**, …)
  - Location of runtime stack (**`%rsp`**)
  - Location of current code control point (**`%rip`**, …)
  - **Status of recent tests (CF, ZF, SF, OF)**



Condition Codes

- Single bit registers
  - **CF** Carry Flag (for unsigned)
  - **SF** Sign Flag (for signed)
  - **ZF** Zero Flag
  - **OF** Overflow Flag (for signed)

- Implicitly set (think of it as side effect) by arithmetic operations

  Example: **`addq`** *Src*,*Dest* ↔ **`t = a+b`**

  CF set if carry out from most significant bit (unsigned overflow)

  ZF set if **`t == 0`**

  SF set if **`t < 0`** (as signed)

  OF set if two's-complement (signed) overflow

  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction
  - **`cmpq`** *Src2*, *Src1*
  - **`cmpq b,a`** like computing **`a-b`** without setting destination

  - **CF set** if carry out from most significant bit (used for unsigned comparisons)
  - **ZF set** if **`a == b`**
  - **SF set** if **`(a-b) < 0`** (as signed)
  - **OF set** if two's-complement (signed) overflow

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

Branches

- Control: Condition codes
- Conditional branches
- Loops

Jumping

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

jX	Description
`jmp`	**Unconditional**
`je`	**Equal / Zero**
`jne`	**Not Equal / Not Zero**
`jg`	**Greater (Signed)**
`jge`	**Greater or Equal (Signed)**
`jl`	**Less (Signed)**
`jle`	**Less or Equal (Signed)**

Conditional Branch Example

```
long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi   # x-y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax   # x-y
    ret
.L4:                     # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax   # y-x
    ret
```

How should we go from conditional branch to assembly code?

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Step 1: Expressing with Goto Code

- Test the condition
- Label the else part
- Place the if part
- Jump to position designated by label

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;




 Done:
    return result;
}
```

Step 1: Expressing with Goto Code

- Test the condition
- Label the else part
- Place the if part
- Jump to position designated by label

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;




 Done:
    return result;
}
```

Step 1: Expressing with Goto Code

- Test the condition
- Label the else part
- Place the if part
- Jump to position designated by label

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;


 Done:
    return result;
}
```

Step 1: Expressing with Goto Code

- Test the condition
- Label the else part
- Place the if part
- Jump to position designated by label

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return resul
}
```

Step 2: Convert Goto Code to Assembly

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi   # x-y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax   # x-y
    ret
.L4:                     # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax   # y-x
    ret
```

Expressing with Goto Code

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi   # x-y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax   # x-y
    ret
.L4:                     # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax   # y-x
    ret
```

Expressing with Goto Code

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi   # x-y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax   # x-y
    ret
.L4:                     # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax   # y-x
    ret
```

Expressing with Goto Code

```
long absdiff
  (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;

    goto Done;
 Else:
    result = y-x;

 Done:
    return result;
}
```

```
absdiff:

    cmpq    %rsi, %rdi   # x-y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax   # x-y
    ret
.L4:                     # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax   # y-x
    ret
```

Summary: Conditional Branch

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
  ntest = !Test;
  if (ntest) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

Branches

- Control: Condition codes
- Conditional branches
- Loops
  - Do-while
  - While
  - for

1. “Do-While” Loop Example

Step 1: Expressing with Goto Code

C Code

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

1. “Do-While” Loop Example

Step 1: Expressing with Goto Code

C Code

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 .loop
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Declare the goto label where the body starts

1. “Do-While” Loop Example

Step 1: Expressing with Goto Code

C Code

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 .loop
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Use conditional branch to
either continue looping or to exit loop

Step 2: Convert Goto Code to Assembly

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

```
    movl    $0, %eax    #  result = 0
.L2:                    # loop:
    movq    %rdi, %rdx
    andl    $1, %edx    #  t = x & 0x1
    addq    %rdx, %rax  #  result += t
    shrq    %rdi        #  x >>= 1
    jne     .L2         #  if (x) goto loop
    ret
```

Register	Use(s)
%rdi	Argument x
%rax	result

1. “Do-While” Loop Example

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

```
    movl    $0, %eax    #  result = 0
.L2:                    # loop:
    movq    %rdi, %rdx
    andl    $1, %edx    #  t = x & 0x1
    addq    %rdx, %rax  #  result += t
    shrq    %rdi        #  x >>= 1
    jne     .L2         #  if (x) goto loop
    ret
```

Register	Use(s)
%rdi	Argument x
%rax	result

1. “Do-While” Loop Example

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;


  if(x) goto loop;
  return result;
}
```

```
    movl    $0, %eax    #  result = 0
.L2:                    # loop:
    movq    %rdi, %rdx
    andl    $1, %edx    #  t = x & 0x1
    addq    %rdx, %rax  #  result += t
    shrq    %rdi        #  x >>= 1
    jne     .L2         #  if (x) goto loop
    ret
```

Register	Use(s)
%rdi	Argument x
%rax	result

1. “Do-While” Loop Example

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;


  if(x) goto loop;
  return result;
}
```

```
   movl    $0, %eax    #  result = 0
.L2:                   # loop:
   movq    %rdi, %rdx
   andl    $1, %edx    #  t = x & 0x1
   addq    %rdx, %rax  #  result += t
   shrq    %rdi        #  x >>= 1
   jne     .L2         #  if (x) goto loop
   ret
```

Register	Use(s)
%rdi	Argument x
%rax	result

1. General “Do-While” Translation

C Code

```
do
  Body
  while (Test);
```

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

- Body:

```
{
  Statement1;
  Statement2;
    …
  Statementn;
}
```

Branches

- Control: Condition codes
- Conditional branches
- Loops
  - Do-while
  - While
  - for

2. General “While” Translation

While version

```
while (Test)
  Body
```

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

Branches

- Control: Condition codes
- Conditional branches
- Loops
  - Do-while
  - While
  - for

3. “For” Loop Form

General Form

```
for (Init; Test; Update)
    Body
```

```
#define WSIZE 8*sizeof(int)
long pcount_for
  (unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
    (x >> i) & 0x1;
  result += bit;
}
```

3. “For” Loop → While Loop

For Version

```
for (Init; Test; Update)
    Body
```

While Version

```
Init;
while (Test) {
    Body
    Update;
}
```

3. For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
Init;
while (Test) {
     Body
     Update;
}
```

```
long pcount_for_while
   (unsigned long x)
{
   size_t i;
   long result = 0;









   }
   return result;
}
```

3. For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
Init;
while (Test) {
     Body
     Update;
}
```

```
long pcount_for_while
   (unsigned long x)
{
   size_t i;
   long result = 0;
   i = 0;




   return result;
}
```

3. For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
init;
while (Test) {
    Body
    Update;
}
```

```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {



  }
  return result;
}
```

3. For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
Init;

while (Test) {

      Body

      Update;

}
```

```
long pcount_for_while
   (unsigned long x)
{
   size_t i;
   long result = 0;
   i = 0;
   while (i < WSIZE)
   {
      unsigned bit =
         (x >> i) & 0x1;
      result += bit;

   }
   return result;
}
```

3. For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
Init;
while (Test) {
     Body
     Update;
}
```

```
long pcount_for_while
  (unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

3': "For" Loop → Do-While Loop

For Version

```
for (Init; Test; Update)
    Body
```

Do-While Version

```
Init;
Loop:
        Body
Update;
Test;
goto Loop;
```

3’: “For” Loop Do-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
Init;
Loop:
        Body

Update;
Test;
goto Loop;
```

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;









 done:
  return result;
}
```

3’: “For” Loop Do-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
Init;
Loop:
        Body
Update;
Test;
goto Loop;
```

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;




 done:
  return result;
}
```

3’: “For” Loop Do-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
Init;
Loop:
        Body
Update;
Test;
goto Loop;
```

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  loop:




 done:
  return result;
}
```

3’: “For” Loop Do-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
Init;
Loop:
        Body
Update;
Test;
goto Loop;
```

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  loop:
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }



 done:
  return result;
}
```

3’: “For” Loop Do-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
Init;
Loop:
        Body
Update;
Test;
goto Loop;
```

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  loop:
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  i++;


 done:
  return result;
}
```

3’: “For” Loop Do-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
Init;

Loop:

        Body


Update;

Test;

goto Loop;
```

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  loop:
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  i++;

  if (i < WSIZE)



 done:
  return result;
}
```

3’: “For” Loop Do-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
Init;
Loop:
        Body
Update;
Test;
goto Loop;
```

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  loop:
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  i++;

  if (i < WSIZE)
    goto loop;


 done:
  return result;
}
```

3’: “For” Loop Do-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

```
Init;
Loop:
        Body
Update;
Test;
goto Loop;
```

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  loop:
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
    i++;

  }

  if (i < WSIZE)
    goto loop;


 done:
  return result;
}
```

Machine Language Summary

- A text representation of machine code
- Three types of operations
  - Arithmetic
  - Data Transfer
  - Control
- Can represent high-level branches
  - Do-While, While, for

Machine Language

CS 3330

Samira Khan
University of Virginia
Feb 2, 2017

While Loop Example

C Code

```
long pcount_while
  (unsigned long x) {
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Do-While Version

```
long pcount_goto_dw
  (unsigned long x) {
  long result = 0;
  if (!x) goto done;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
 done:
  return result;
}
```

Initial conditional guards entrance to loop