Lecture 2
Larry Rudolph
Computer System Organization

- CPU
  - MMU
  - Memory
  - Controller
- Local Bus
- Interface
- High-Speed I/O Bus
  - NIC
  - Controller
  - Bridge
  - Frame Buffer
- Low-Speed I/O Bus
  - CD-ROM
  - USB
CPU Organization

• Instruction Set Architecture (ISA)
  Defines:
  • the state visible to the programmer
    • registers and memory
  • the instruction that operate on the state

• ISA typically divided into 2 parts
  • User ISA
    • Primarily for computation
  • System ISA
    • Primarily for system resource management
User ISA - State

User Virtual Memory

Special-Purpose Registers
- Program Counter
- Condition Codes

General-Purpose Registers
- Reg 0
- Reg 1
- Reg n-1

Floating Point Registers
- FP 0
- FP 1
- FP n-1
System ISA

• Privilege Levels

• Control Registers

• Traps and Interrupts
  • Hardcoded Vectors
  • Dispatch Table

• System Clock

• MMU
  • Page Tables
  • TLB

• I/O Device Access
What is a Virtual Machine Monitor?

• Classic Definition (Popek and Goldberg ’74)

A virtual machine is taken to be an efficient, isolated duplicate of the real machine. We explain these notions through the idea of a virtual machine monitor (VMM). See Figure 1. As a piece of software a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.
CPU Virtualization

Example: CPUState

```c
static struct {
    uint32 GPR[16];
    uint32 LR;
    uint32 PC;
    int IE;
    int IRQ;
} CPUState;

void CPU_CLI(void)
{
    CPUState.IE = 0;
}

void CPU_STI(void)
{
    CPUState.IE = 1;
}
```

- Goal for CPU virtualization techniques
  - Process normal instructions as fast as possible
  - Forward privileged instructions to emulation routines
Virtualizing the System ISA

• Hardware needed by monitor
  • Ex: monitor must control real hardware interrupts

• Access to hardware would allow VM to compromise isolation boundaries
  • Ex: access to MMU would allow VM to write any page

• So…
  • All access to the virtual System ISA by the guest must be emulated by the monitor in software.
  • System state kept in memory.
  • System instructions are implemented as functions in the monitor.
Trap and Emulate

Guest OS + Applications

Virtual Machine Monitor

Unprivileged

Privileged

- Page Fault
- Undef Instr
- vIRQ

- MMU Emulation
- CPU Emulation
- I/O Emulation
Trap and Emulate

All traps caught by VMM

Applications

Guest OS

Virtual Machine Monitor

VMM updates Guest Virtual Machine State

VMM returns to Guest OS to let it handle the trap
“Strictly Virtualizable”

A processor or mode of a processor is strictly virtualizable if, when executed in a lesser privileged mode:

• all instructions that access privileged state trap

• all instructions either trap or execute identically
Issues with Trap and Emulate

• Not all architectures support it

• Trap costs may be high

• VMM consumes a privilege level
  • Need to virtualize the protection levels
So, What is the Solution?

• Can we let these privileged instructions execute while in user mode?
  • No … emulation will be wrong
  • example, disable interrupts operation executes but does not disable them. What is the big deal?
    • Race conditions. What else?
  • But how does disabling virtual interrupts help? Are interrupts really disabled?
    • Who knows. VMM will not pass the interrupt to the guest os while the guest has its interrupts disabled.

• Possible Solutions:
  • Somehow make sure these instructions trap
  • Execute these instructions in privileged mode —— does not work
  • Don’t execute them —— Huh?
Solution: Directly modify the emulated machine state

- Replace the instruction with trap
- Replace the instruction with call to emulation procedure
- Replace the instruction with emulation code
- In other words
  - Translate these difficult instructions to something else
  - And while we’re at it, maybe do it for all sensitive instructions.
- “Scan before execute” also known as “Binary Translation”
Binary Translation

- Guest Code
- Translator
- Translation Cache
- Callouts
- CPU Emulation Routines

Callouts
- TC Index
User-level ISA (instructions)

Typical Instruction Pipeline

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Registers</th>
<th>Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For each of the above groups, typical examples are:

**Integer**
- Add
- Sub
- And
- Compare
- ...

**Memory**
- Load byte
- Load Word
- Store Multiple
- Push
- ...

**Control Flow**
- Jump
- Jump equal
- Call
- Return
- ...

**Floating Point**
- Add single
- Mult. double
- Sqrt double
- ...

Instruction Groupings
System-Level ISA groups

- Privilege Levels
- Control Registers
- Traps and Interrupts
  - Hardcoded Vectors
  - Dispatch Table
- System Clock
- MMU
  - Page Tables
  - TLB
- I/O Device Access
Basic Block

- **cli** instruction “clears” the interrupt flag
  - disables interrupts
- **sti** instruction “sets” the interrupt flag
  - enables interrupts
- **cr3** control register 3 points to the page table directory
  - privileged instruction
Call emulation routine

```
Guest Code

vPC
mov ebx, eax
cli
and ebx, ~0xfff
mov ebx, cr3
sti
ret

Translation Cache

mov ebx, eax

call HANDLE_CLI
and ebx, ~0xfff
mov [CO_ARG], ebx
call HANDLE_CR3
call HANDLE_STI
jmp HANDLE_RET

start
```
Execute Emulation Code

Guest Code

```plaintext
mov ebx, eax
cli
and ebx, ~0xfff
mov ebx, cr3
sti
ret
```

Translation Cache

```plaintext
mov ebx, eax
mov [CPU_IE], 0
and ebx, ~0xfff
mov [CO_ARG], ebx
```

```plaintext
mov [CPU_IE], 1
```

```plaintext
test [CPU_IRQ], 1
jne
```

```plaintext
call HANDLE_INTS
jmp HANDLE_RET
```

```plaintext
start
mov ebx, cr3
sti
ret
```
Basic Translator

```c
void BT_Run(void)
{
    CPUState.PC = _start;
    BT_Continue();
}

void BT_Continue(void)
{
    void *tcpc;
    tcpc = BTFindBB(CPUState.PC);
    if (!tcpc) {
        tcpc = BTTranslate(CPUState.PC);
    }
    RestoreRegsAndJump(tcpc);
}

void *BTTranslate(uint32 pc)
{
    void *start = TCTop;
    uint32 TCPC = pc;
    while (1) {
        inst = Fetch(TCPC);
        TCPC += 4;
        if (IsPrivileged(inst)) {
            EmitCallout();
        } else if (IsControlFlow(inst)) {
            EmitEndBB();
            break;
        } else {
            /* ident translation */
            EmitInst(inst);
        }
    }
    return start;
}
```
Basic Translator

```c
void BT_CalloutSTI(BTSavedRegs regs)
{
    CPUState.PC = BTFindPC(regs.tcpc);
    CPUState.GPR[] = regs.GPR[];

    CPU_STI();

    CPUState.PC += 4;

    if (CPUState.IRQ && CPUState.IE) {
        CPUVector();
        BT_Continue();
        /* NOT_REACHED */
    }

    return;
}
```
Controlling the Control Flow

Guest Code

```
vePC

<table>
<thead>
<tr>
<th></th>
<th>Translation Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>test eax, 1</td>
<td>test eax, 1</td>
</tr>
<tr>
<td>jeq</td>
<td>jeq</td>
</tr>
<tr>
<td>add ebx, 18</td>
<td>call END_BB</td>
</tr>
<tr>
<td>mov ecx, [ebx]</td>
<td>call END_BB</td>
</tr>
<tr>
<td>mov [ecx], eax</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>
```

```
vEPC

test  eax, 1
jeq
add   ebx, 18
mov   ecx, [ebx]
mov   [ecx], eax
ret
```
Controlling the Control Flow

test eax, 1
jeq
add ebx, 18
mov ecx, [ebx]
mov [ecx], eax
ret

test eax, 1
jeq
call END_BB
call END_BB
add ebx, 18
mov ecx, [ebx]
mov [ecx], eax
call HANDLE_RET

eax == 0
Controlling Flow

Guest Code

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>test eax, 1</td>
</tr>
<tr>
<td>jeq</td>
</tr>
<tr>
<td>add ebx, 18</td>
</tr>
<tr>
<td>mov ecx, [ebx]</td>
</tr>
<tr>
<td>mov [ecx], eax</td>
</tr>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>

Translation Cache

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>test eax, 1</td>
</tr>
<tr>
<td>jeq</td>
</tr>
<tr>
<td>jmp</td>
</tr>
<tr>
<td>call END_BB</td>
</tr>
<tr>
<td>add ebx, 18</td>
</tr>
<tr>
<td>mov ecx, [ebx]</td>
</tr>
<tr>
<td>mov [ecx], eax</td>
</tr>
<tr>
<td>call HANDLE_RET</td>
</tr>
</tbody>
</table>

eax == 0
Controlling Control Flow

Guest Code

- `test eax, 1`
- `jeq`
- `add ebx, 18`
- `mov ecx, [ebx]`
- `mov [ecx], eax`
- `ret`

Translation Cache

- `test eax, 1`
- `jeq`
- `jmp`
- `call END_BB`
- `add ebx, 18`
- `mov ecx, [ebx]`
- `mov [ecx], eax`
- `call HANDLE_RET`
- `mov [ecx], eax`
- `call HANDLE_RET`

`eax == 1`
Controlling

Guest Code

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>Translation Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>test eax, 1</td>
<td>test eax, 1</td>
</tr>
<tr>
<td>jeq</td>
<td>jeq</td>
</tr>
<tr>
<td>add ebx, 18</td>
<td>jmp</td>
</tr>
<tr>
<td>mov ecx, [ebx]</td>
<td>mov ecx, [ebx]</td>
</tr>
<tr>
<td>mov [ecx], eax</td>
<td>mov [ecx], eax</td>
</tr>
<tr>
<td>ret</td>
<td>call HANDLE_RET</td>
</tr>
</tbody>
</table>
```

eax == 1
Issues with Binary Translation

• Translation cache management

• PC synchronization on interrupts

• Self-modifying code
  • Notified on writes to translated guest code

• Protecting VMM from guest
Hybrid Approach

- Binary Translation for the Kernel
- Direct Execution (Trap-and-emulate) for the User
- U.S. Patent 6,397,242
Other Uses for Binary Translation

• Cross ISA translators
  • Apple’s Rosetta – PowePC to Intel, Digital FX!32

• Optimizing translators
  • HP Dynamo

• High level language byte code translators
  • Java
  • .NET/CLI
Instruction Set Architectures

• CISC: Complex Instruction Set Computer
  • New machines would add more “interesting” inst

• RISC: Reduced Instruction Set Computer
  • Execution of several inst faster than some single CISC inst
  • Load-Store between memory and reg, and ops between registers only

• Intel’s x86, Intel 64, AMD64 are the dominant ones

• ARM has 4 different ISA’s in same architecture

• Many others have died: PPC, MIPS, ….

• Several efforts at open source ISA’s
  • RISC-V
Quick Overview of ISA’s

Instruction Set Architecture and its Implications

A. Moshovos (c)

Based on H&P CA: AQA
Some Figures and Tables taken directly from the book
These are not meant to be slides but notes
Some material from notes by Hill, Wood, Sohi and Smith

Fall 2005
© A. Moshovos (Univ. of Toronto)
Do ISA’s Matter?

- Provided have the right coverage, no since memory hierarchy is biggest affect on performance


- Herterogeneous-ISA Chip Multiprocessor, can give 20% performances and 30% energy improvement

  - Harnessing ISA Diversity (http://cseweb.ucsd.edu/~asvenkat/papers/isca2014.pdf)
ISA Examination
Intel X86 and Intel 64

• Very very complex

• Variable length instructions

• Tons of modes

• byte, half word, word, double word, quad word, ??

• Many generations, variants

• Difficult to decode due to expanding set of insts
## CodeTable 1/2

### TRANSFER

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Flags</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>Move (copy)</td>
<td>MOV Dest,Source</td>
<td>Dest:=Source</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STC</td>
<td>Set Carry</td>
<td>STC</td>
<td>CF:=1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLC</td>
<td>Clear Carry</td>
<td>CLC</td>
<td>CF:=0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMC</td>
<td>Complement Carry</td>
<td>CMC</td>
<td>CF:=¬CF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>±</td>
</tr>
<tr>
<td>STD</td>
<td>Set Direction</td>
<td>STD</td>
<td>DF:=1 (string op's downwards)</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLD</td>
<td>Clear Direction</td>
<td>CLD</td>
<td>DF:=0 (string op's upwards)</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STI</td>
<td>Set Interrupt</td>
<td>STI</td>
<td>IF:=1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLI</td>
<td>Clear Interrupt</td>
<td>CLI</td>
<td>IF:=0</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSH</td>
<td>Push onto stack</td>
<td>PUSH Source</td>
<td>DEC SP, [SP]:=Source</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSHF</td>
<td>Push flags</td>
<td>PUSHF</td>
<td>O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSHA</td>
<td>Push all general registers</td>
<td>PUSHA</td>
<td>AX, CX, DX, BX, SP, BP, SI, DI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POP</td>
<td>Pop from stack</td>
<td>POP Dest</td>
<td>Dest:=[SP], INC SP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>POPF</td>
<td>Pop flags</td>
<td>POPF</td>
<td>O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL</td>
<td>±</td>
<td>±</td>
<td>±</td>
<td>±</td>
<td>±</td>
<td>±</td>
<td>±</td>
<td>±</td>
<td></td>
<td></td>
</tr>
<tr>
<td>POPA</td>
<td>Pop all general registers</td>
<td>POPA</td>
<td>DI, SI, BP, SP, BX, DX, CX, AX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBW</td>
<td>Convert byte to word</td>
<td>CBW</td>
<td>AX:=AL (signed)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CWD</td>
<td>Convert word to double</td>
<td>CWD</td>
<td>DX:AX:=AX (signed)</td>
<td>±</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CWDE</td>
<td>Conv word extended double</td>
<td>CWDE</td>
<td>EAX:=AX (signed)</td>
<td>±</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* **i** for more information see instruction specifications

**Flags: ±=affected by this instruction  ?=undefined after this instruction**
### ARITHMETIC

| Name | Comment | Code | Operation | O | D | I | T | S | Z | A | P | C |
|------|---------|------|-----------|---|---|---|---|---|---|---|---|---|---|
| ADD  | Add     | ADD Dest, Source | Dest:=Dest+Source | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| ADC  | Add with Carry | ADC Dest, Source | Dest:=Dest+Source+CF | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| SUB  | Subtract | SUB Dest, Source | Dest:=Dest-Source | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| SBB  | Subtract with borrow | SBB Dest, Source | Dest:=(Dest+Source)+CF | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| MUL  | Multiply (unsigned) | MUL Op | Op=byte: AX:=AL*Op if AH=0 | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| MUL  | Multiply (unsigned) | MUL Op | Op=word: DX:=AX:AX*Op if DX=0 | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| MUL 386 | Multiply (unsigned) | MUL Op | Op=doubelw.: EAX:=EAX*Op if EDX=0 | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| IMUL | Signed Integer Multiply | IMUL Op | Op=word: AX:=DX:AX*Op if AX sufficient | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| IMUL 386 | Signed Integer Multiply | IMUL Op | Op=doubelw.: EAX:=EAX*Op if EAX sufficient | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| INC  | Increment | INC Op | Op:=Op+1 (Carry not affected) | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| DEC  | Decrement | DEC Op | Op:=Op-1 (Carry not affected) | ± | ± | ± | ± | ± | ± | ± | ± | ± |
| CMP  | Compare | CMP Op1,Op2 | Op1-Op2 | ± | ± | ± | ± | ± | ± | ± | ± | ± |

---

### LOGIC

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEG</td>
<td>Negate (two-complement)</td>
<td>NEG Op</td>
<td>Op:=0-Op if Op=0 then CF:=0 else CF:=1</td>
<td>±</td>
</tr>
<tr>
<td>NOT</td>
<td>Invert each bit</td>
<td>NOT Op</td>
<td>Op:=~Op (invert each bit)</td>
<td>±</td>
</tr>
<tr>
<td>AND</td>
<td>Logical and</td>
<td>AND Dest, Source</td>
<td>Dest:=Dest+Source</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>Logical or</td>
<td>OR Dest, Source</td>
<td>Dest:=Dest+Source</td>
<td>0</td>
</tr>
<tr>
<td>XOR</td>
<td>Logical exclusive or</td>
<td>XOR Dest, Source</td>
<td>Dest:=Dest(exor) Source</td>
<td>0</td>
</tr>
<tr>
<td>SHL</td>
<td>Shift logical left (= SAL)</td>
<td>SHL Op, Quantity</td>
<td>Dest:=Dest+Source</td>
<td>0</td>
</tr>
<tr>
<td>SHR</td>
<td>Shift logical right</td>
<td>SHR Op, Quantity</td>
<td>Dest:=Dest+Source</td>
<td>0</td>
</tr>
</tbody>
</table>
## Flags:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>Carry</td>
</tr>
<tr>
<td>D</td>
<td>Direction</td>
</tr>
<tr>
<td>I</td>
<td>Interrupt</td>
</tr>
<tr>
<td>T</td>
<td>Trap</td>
</tr>
</tbody>
</table>

## General Registers:

### Accumulator
- EAX 386
- AX
- AL

### Data mul, div, IO
- EDX 386
- DX
- DL

### Count loop, shift
- ECX 386
- CX
- CL

### BaseX data ptr
- EBX 386
- BX
- BL

## JUMPS (flags remain unchanged)

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>Call subroutine</td>
<td>CALL Proc</td>
<td>RET</td>
</tr>
<tr>
<td>JMP</td>
<td>Jump</td>
<td>JMP Dest</td>
<td></td>
</tr>
<tr>
<td>JE</td>
<td>Jump if Equal</td>
<td>JE Dest</td>
<td>JNE</td>
</tr>
<tr>
<td>JZ</td>
<td>Jump if Zero</td>
<td>JZ Dest</td>
<td>JNZ</td>
</tr>
<tr>
<td>JCEXZ</td>
<td>Jump if CX Zero</td>
<td>JCEXZ Dest</td>
<td>JECXZ</td>
</tr>
<tr>
<td>JP</td>
<td>Jump if Parity (Parity Even)</td>
<td>JP Dest</td>
<td>JNP</td>
</tr>
<tr>
<td>JPE</td>
<td>Jump if Parity Even</td>
<td>JPE Dest</td>
<td>JPO</td>
</tr>
</tbody>
</table>

## JUMPS Signed (Integer)

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JA</td>
<td>Jump if Above</td>
<td>JA Dest</td>
<td>(≡ JA)</td>
</tr>
<tr>
<td>JAE</td>
<td>Jump if Above or Equal</td>
<td>JAE Dest</td>
<td>(≡ JNE + JNC)</td>
</tr>
<tr>
<td>JB</td>
<td>Jump if Below</td>
<td>JB Dest</td>
<td>(≡ JNE + JC)</td>
</tr>
<tr>
<td>JBE</td>
<td>Jump if Below or Equal</td>
<td>JBE Dest</td>
<td>(≡ JNC)</td>
</tr>
<tr>
<td>JNA</td>
<td>Jump if not Above</td>
<td>JNA Dest</td>
<td>(≡ JE)</td>
</tr>
<tr>
<td>JNBE</td>
<td>Jump if not Above or Equal</td>
<td>JNBE Dest</td>
<td>(≡ JE + JNC)</td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if not Below</td>
<td>JNB Dest</td>
<td>(≡ JNE + JC)</td>
</tr>
<tr>
<td>JNBE</td>
<td>Jump if not Below or Equal</td>
<td>JNBE Dest</td>
<td>(≡ JA)</td>
</tr>
<tr>
<td>JC</td>
<td>Jump if Carry</td>
<td>JC Dest</td>
<td>(≡ JA)</td>
</tr>
<tr>
<td>JNC</td>
<td>Jump if no Carry</td>
<td>JNC Dest</td>
<td>(≡ JA)</td>
</tr>
</tbody>
</table>

## JUMPS Unsigned (Cardinal)

<table>
<thead>
<tr>
<th>Name</th>
<th>Comment</th>
<th>Code</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JA</td>
<td>Jump if Above</td>
<td>JA Dest</td>
<td>(≡ JA)</td>
</tr>
<tr>
<td>JAE</td>
<td>Jump if Above or Equal</td>
<td>JAE Dest</td>
<td>(≡ JNE + JNC)</td>
</tr>
<tr>
<td>JB</td>
<td>Jump if Below</td>
<td>JB Dest</td>
<td>(≡ JNE + JC)</td>
</tr>
<tr>
<td>JBE</td>
<td>Jump if Below or Equal</td>
<td>JBE Dest</td>
<td>(≡ JNC)</td>
</tr>
<tr>
<td>JNA</td>
<td>Jump if not Above</td>
<td>JNA Dest</td>
<td>(≡ JE)</td>
</tr>
<tr>
<td>JNBE</td>
<td>Jump if not Above or Equal</td>
<td>JNBE Dest</td>
<td>(≡ JE + JNC)</td>
</tr>
<tr>
<td>JNB</td>
<td>Jump if not Below</td>
<td>JNB Dest</td>
<td>(≡ JNE + JC)</td>
</tr>
<tr>
<td>JNBE</td>
<td>Jump if not Below or Equal</td>
<td>JNBE Dest</td>
<td>(≡ JA)</td>
</tr>
</tbody>
</table>

## General Registers:

### Accumulator
- EAX 386
- AX
- AL

### Data mul, div, IO
- EDX 386
- DX
- DL

### Count loop, shift
- ECX 386
- CX
- CL

### BaseX data ptr
- EBX 386
- BX
- BL

## Flags:

- O: Carry
- D: Direction
- I: Interrupt
- T: Trap
- C: Carry

### Control Flags (how instructions are carried out):

- D: Direction: 1 = string op's process down from high to low address
- I: Interrupt: whether interrupts can occur. 1= enabled
- T: Trap: single step for debugging

---

CSCI-GA.3033-010 (Fall 2014) Virtualization Technology Larry Rudolph
ISA RISC-V

- Fairly new (a few years), Developed at Berkeley
- Very clean, functional, expandable, understandable
- FPGA implementations
The RISC-V Instruction Set Manual
Volume I: User-Level ISA
Version 2.0

Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanović
CS Division, EECS Department, University of California, Berkeley
{waterman|yunsup|pattrsn|krste}@eecs.berkeley.edu
May 6, 2014

This document is also available as Technical Report
UCB/EECS-2014-54.

Copyright © 2010–2014, The Regents of the University of California. All rights reserved.

16-bit (aa ≠ 11)

32-bit (bbb ≠ 111)

48-bit

64-bit

(80+16*nmm)-bit, nmm ≠ 111

Reserved for ≥320-bits

Figure 1.1: RISC-V instruction length encoding.

Although an instruction encoding space could be of any size, adopting a smaller set of common
sizes simplifies packing independently developed extensions into a single global encoding. Table 9.1
gives the suggested sizes for RISC-V.

<table>
<thead>
<tr>
<th>Size</th>
<th>Usage</th>
<th># Available in standard instruction length</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit</td>
<td>Quadrant of compressed 16-bit encoding</td>
<td>3</td>
</tr>
<tr>
<td>32-bit</td>
<td>Minor opcode in base 32-bit encoding</td>
<td>2^8</td>
</tr>
<tr>
<td>48-bit</td>
<td>Major opcode in base 32-bit encoding</td>
<td>2^27</td>
</tr>
<tr>
<td>64-bit</td>
<td>Quadrant of base 32-bit encoding</td>
<td>2^27</td>
</tr>
<tr>
<td>32-bit</td>
<td>Minor opcode in 48-bit encoding</td>
<td>2^25</td>
</tr>
<tr>
<td>40-bit</td>
<td>Major opcode in 48-bit encoding</td>
<td>2^20</td>
</tr>
<tr>
<td>48-bit</td>
<td>Quadrant of 48-bit encoding</td>
<td>2^17</td>
</tr>
<tr>
<td>64-bit</td>
<td>Sub-minor opcode in 64-bit encoding</td>
<td>2^12</td>
</tr>
<tr>
<td>48-bit</td>
<td>Minor opcode in 64-bit encoding</td>
<td>2^12</td>
</tr>
<tr>
<td>52-bit</td>
<td>Major opcode in 64-bit encoding</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 9.1: Suggested standard RISC-V instruction encoding space sizes.
Virtualization Technology

The only di

2.3 Immediate Encoding Variants

2.2 Base Instruction Formats

Copyright

Although more complex implementations might have separate adders for branch and jump

In practice, most immediates are either small or require all XLEN bits. We chose an asym-

Static instruction bits used to construct their value. Sign extension always uses inst[31].

Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the

31 30 25 24 20 19 15 14 12 11 8 7 6 0

funct7 rs2 rs1 funct3 rd opcode 
imm[11:0] rs1 funct3 rd opcode I-type
imm[31:12] rd opcode U-type

Figure 2.2: RISC-V base instruction formats.

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0

funct7 rs2 rs1 funct3 rd opcode R-type

Figure 2.3: RISC-V base instruction formats showing immediate variants.

31 30 20 19 12 11 10 5 4 1 0


Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the

instruction bits used to construct their value. Sign extension always uses inst[31].
QEMU

- ISA Emulator
  - For many different ISA’s — different source/target
- Emulation Platform
- Virtualization Platform — binary translation
- Please install riscv version