Disclaimer: Many slides of this lecture are based on the slides of authors of the textbook from Elsevier. All copyrights reserved.
Emulation

- **Why** are we studying it here?
  - Techniques that can be applied to any type of VM.

- **Definition**: The process of **implementing the interface and functionality** of one system on a system with different interface and functionality.

- **Science**: How to do it?

- **Art**: How to do it well? → performance
Key VM Technologies

- **Emulation**: binary in one ISA is executed on processor supporting a different ISA
- **Dynamic Optimization**: binary is improved for higher performance
  - May be done as part of emulation
  - May optimize same ISA (no emulation needed)
Definitions

• **Guest**
  – Environment that is being supported by underlying platform

• **Host**
  – Underlying platform that provides guest environment
Definitions

• **Source ISA or binary**
  – Original instruction set or binary
    I.e. the instruction set to be emulated

• **Target ISA or binary**
  – Instruction set being executed by processor performing emulation
    I.e. the underlying instruction set
  Or the binary that is actually executed

Source/Target refer to ISAs; Guest/Host refer to platforms.

Warning: You may not find standard definition in literature ... Unfortunately!
How to do emulation?

Interpretation
- Fetch source instruction
- Analyze it
- Perform the required operation

Binary Translation
- block of source instructions → block of target instructions
- Save for repeated use
Interpretation
Interpreters

• HLL Interpreters have a very long history
  – Lisp
  – Perl
  – Python

• Binary interpreters use many of the same techniques
  – Often simplified
  – Performance tradeoffs
Interpreter State

• Hold complete source state in interpreter’s data memory
while (!halt && !interrupt) {
    inst = code[PC];
    opcode = extract(inst, 31, 6);
    switch(opcode) {
        case LoadWordAndZero: LoadWordAndZero(inst);
        case ALU: ALU(inst);
        case Branch: Branch(inst);
        . . .
    }
}

LoadWordAndZero(inst){
    RT = extract(inst, 25, 5);
    RA = extract(inst, 20, 5);
    displacement =
        extract(inst, 15, 16);
    if (RA == 0) source = 0;
    else source = regs[RA];
    address = source + displacement;
    regs[RT] =
        (data[address] << 32) >> 32;
    PC = PC + 4;
}

ALU(inst){
    RT = extract(inst, 25, 5);
    RA = extract(inst, 20, 5);
    RB = extract(inst, 15, 5);
    source1 = regs[RA];
    source2 = regs[RB];
    extended_opcode =
        extract(inst, 10, 10);
    switch(extended_opcode) {
        case Add: Add(inst);
        case AddCarrying: AddCarrying(inst);
        case AddExtended: AddExtended(inst);
        . . .
    }
    PC = PC + 4;
}
while (!halt && !interrupt) {
  inst = code[PC];
  opcode = extract(inst, 31, 6);
  switch (opcode) {
    case LoadWordAndZero: LoadWordAndZero(inst);
    case ALU: ALU(inst);
    case Branch: Branch(inst);
    ...
  }
}

Instruction function list

LoadWordAndZero(inst) {
  RT = extract(inst, 25, 5);
  RA = extract(inst, 20, 5);
  displacement =
    extract(inst, 15, 16);
  if (RA == 0) source = 0;
  else source = regs[RA];
  address = source + displacement;
  regs[RT] =
    (data[address] << 32) >> 32;
  PC = PC + 4;
}

ALU(inst) {
  RT = extract(inst, 25, 5);
  RA = extract(inst, 20, 5);
  RB = extract(inst, 15, 5);
  source1 = regs[RA];
  source2 = regs[RB];
  extendedOpcode =
    extract(inst, 10, 10);
  switch(extendedOpcode) {
    case Add: Add(inst);
    case AddCarrying:
      AddCarrying(inst);
    case AddExtended:
      AddExtended(inst);
    ...
  }
  PC = PC + 4;
}

Branches are expensive!
Decode-Dispatch: Efficiency

- **Decode-Dispatch Loop**
  - Mostly serial code
  - *Condition check and branch in the main loop*
  - *Case statement (hard-to-predict indirect branch)*
  - Call to function routine
  - Return
Version 2: Threaded Interpretation

- Remove decode-dispatch jump inefficiency
  (By getting rid of decode-dispatch loop)
- Copy decode-dispatch into the function routines
  - Source code “threads” together function routines
Version 2: Threaded Interpretation

LoadWordAndZero:

RT = extract(inst,25,5);
RA = extract(inst,20,5);
displacement = extract(inst,15,16);
if (RA == 0) source = 0;
else source = regs(RA);
address = source + displacement;
regs(RT) =
    (data(address) « 32) >> 32;
PC = PC +4;
If (halt || interrupt) goto exit;
ins = code[PC];
opcode = extract(inst,31,6)
extended_opcode =
    extract(inst,10,10);
routine =
    dispatch[opcode,extended_opcode];
goto *routine;

Add:

RT = extract(inst,25,5);
RA = extract(inst,20,5);
RB = extract(inst,15,5);
soure1 = regs(RA);
soure2 = regs[RB];
sum = source1 + source2;
regs[RT] = sum;
PC = PC +4;
If (halt || interrupt)
    goto exit;
inst = code[PC];
opcode = extract(inst,31,6)
extended_opcode =
    extract(inst,10,10);
routine =
    dispatch[opcode,extended_opcode];
goto *routine;
Version 2: Threaded Interpretation

LoadWordAndZero:

RT = extract(inst, 25, 5);
RA = extract(inst, 20, 5);
displacement = extract(inst, 15, 16);
if (RA == 0) source = 0;
else source = regs(RA);
address = source + displacement;
regs(RT) =
    (data(address) << 32) >> 32;
PC = PC + 4;
If (halt || interrupt) goto exit;
inst = code[PC];
opcode = extract(inst, 31, 6)
extended_opcode =
    extract(inst, 10, 10);
routine =
    dispatch[opcode, extended_opcode];
goto *routine;

Add:

RT = extract(inst, 25, 5);
RA = extract(inst, 20, 5);
RB = extract(inst, 15, 5);
source1 = regs(RA);
source2 = regs[RB];
sum = source1 + source2;
regs[RT] = sum;
PC = PC + 4;
If (halt || interrupt)
    goto exit;
inst = code[PC];
opcode = extract(inst, 31, 6)
extended_opcode =
    extract(inst, 10, 10);
routine =
    dispatch[opcode, extended_opcode];
goto *routine;
Comparison

source code

"data" accesses

dispatch loop

Decode-dispatch

interpreter routines

Threaded

source code

interpreter routines
Evaluation

+ By appending potion of the dispatch code to the end of each instruction routine, we remove 3 out of the 4 branches!

- routine = dispatch[opcode,extended_opcode]; This dispatch table that contains the address of the next subroutine is still an expensive operation (memory access + indirect jump). This is why version 2 is called *indirect threaded interpreter*. 
Version 3: Predecoding

- **Main idea**: Scan static instructions and convert them to internal pre-decoded form
  - Separate the opcode, operands, etc.
- **Why it matters?** Reduces shift/masks significantly
  - Because we do them once and not every time the instruction is encountered.

Example:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>lwz</td>
<td>07</td>
<td>1 2 08</td>
</tr>
<tr>
<td>r1, 8(r2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>08</td>
<td>3 1 03</td>
</tr>
<tr>
<td>r3, r3,r1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stw</td>
<td>37</td>
<td>3 4 00</td>
</tr>
<tr>
<td>r3, 0(r4)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(load word and zero)

(add)

(store word)
Version 3: Predecoding

LoadWordAndZero:

RT = extract(inst, 25, 5);
RA = extract(inst, 20, 5);
displacement =
extract(inst, 15, 16);
if (RA == 0) source = 0;
else source = regs(RA);
address = source + displacement;
regs(RT) =
(data(address) << 32) >> 32;
PC = PC + 4;
If (halt || interrupt) goto exit;
inst = code[PC];
opcode = extract(inst, 31, 6)
extended_opcode =
extract(inst, 10, 10);
routine =
dispatch[opcode, extended Opcode];
goto *routine;

struct instruction {
    unsigned long op;
    unsigned char dest;
    unsigned char src1;
    unsigned int src2;
} code [CODE_SIZE]

Load Word and Zero:
RT = code[TPC].dest;
RA = code[TPC].src1;
displacement = code[TPC].src2;
if (RA == 0) source = 0;
else source = regs[RA];
address = source + displacement;
regs[RT] = (data[address] << 32) >> 32;
SPC = SPC + 4;
TPC = TPC + 1;
If (halt || interrupt) goto exit;
opcode = code[TPC].op;
routine = dispatch[opcode];
goto *routine;

TPC: Target PC ← used to access code[]
SPC: Source PC ← Architected PC
Version 3: Evaluation

- Still we have to access dispatch[] table
  - routine = dispatch[opcode]
Version 4: Direct Threaded Interpretation

- Eliminate indirection through dispatch table
  - Replace opcodes with PCs of routines
- Becomes dependent on locations of interpreter routines
  - Limits portability of interpreter

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>001048d0</td>
<td>1 2 08</td>
<td></td>
<td>(load word and zero)</td>
</tr>
<tr>
<td>00104800</td>
<td>3 1 03</td>
<td></td>
<td>(add)</td>
</tr>
<tr>
<td>00104910</td>
<td>3 4 00</td>
<td></td>
<td>(store word)</td>
</tr>
</tbody>
</table>
struct instruction {
    unsigned long op;
    unsigned char dest;
    unsigned char src1;
    unsigned int src2;
} code [CODE_SIZE]

Load Word and Zero:
RT = code[TPC].dest;
RA = code[TPC].src1;
displacement = code[TPC].src2;
if (RA == 0) source = 0;
else source = regs[RA];
address = source +
    displacement;
regs[RT] =
    (data[address]<< 32) >> 32;
SPC = SPC + 4;
TPC = TPC + 1;
If (halt || interrupt) goto exit;
opcode = code[TPC].op;
routine = dispatch[opcode];
goto *routine;

Load Word and Zero:
RT = code[TPC].dest;
RA = code[TPC].src1;
displacement = code[TPC].src2;
if (RA == 0) source = 0;
else source = regs[RA];
address = source +
    displacement;
regs[RT] =
    (data[address]<< 32) >> 32;
SPC = SPC + 4;
TPC = TPC + 1;
If (halt || interrupt) goto exit;
routine = dispatch[opcode];
goto *routine;
Version 4: Direct Threaded Interpretation

source code

pre-decoder

intermediate code

interpreter routines
Useful technique:
Pointer to Functions in C

Example:
```c
char somefunction(int x, float y)
{
    some code in here ....
}
```

To declare a pointer to it, we need to do three things:

1. Declare a pointer to a function that returns char and gets int and float:
   ```c
   char (* ptr)(int, float);
   ```

2. Make this pointer point to the above function (or any other function that accepts int and float and return char):
   ```c
   ptr = &somefunction;
   ```

3. Use that pointer to call the function.
   ```c
   char k;
   int x = 5;
   float y = 5.75;
   k = (*ptr)(x, y);
   ```
Challenge: Interpreting CISC ISA

- Why challenging? Instructions do not have regular format.
- IA-32 instruction format:

<table>
<thead>
<tr>
<th>Prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to 4</td>
<td>optional</td>
<td>optional</td>
<td>optional</td>
<td>0,1,2,4 bytes</td>
<td>0,1,2,4 bytes</td>
</tr>
</tbody>
</table>

For string instructions and/or address overriding:

- For some of opcodes:
  - indicates an address mode and a register
  - For: scale, Index reg, Base reg
Interpreting x86

Maintain general purpose instruction template

- **Phase 1**: Parse and fill template
- **Phase 2**: dispatch/execute

General Decode (fill-in instruction structure)

Dispatch

Inst. 1 specialized routine

Inst. 2 specialized routine

Inst. n specialized routine
The Instruction Template

struct IA-32instr {
    unsigned short opcode, prefixmask;
    char ilen; // instruction length.
    InterpreterFunctionPointer execute;
    struct {
        //general addr. computation: [Rbase+(Rindex << shmt)+ disp.]
        char mode; // addressing mode, including register operand.
        char Rbase; // base address register
        char Rindex; // index register
        char shmt; // index scale factor
        long displacement;
    } operandRM;
    struct {
        char mode; // either register or immediate.
        char regname; // register number
        long immediate;// immediate value
    } operandRI;
} instr;
The Main Loop

```c
void cpu_loop()
{
    while (!halt) {
        instr = IA-32_FetchDecode(PC);
        if (!IA-32_string_instruction) {
            instr.execute();
        }
        else {
            while (need_to_repeat(instr.prefixmask))
                instr.execute();
            handle_asyn_event(); // i.e. an interrupt
        }
    }
    PC = PC + instr.ilen;
    handle_asyn_event();
}
```
Make the common case fast

common case:
- no prefix
- 1 byte opcode
- simple operad specifiers (regs)

Dispatch on first byte

Simple Inst. 1 specialized routine

Simple Inst. m specialized routine

Complex Inst. m+1 specialized routine

Complex Inst. n specialized routine

Prefix set flags

Shared Routines
Threaded x86 Interpreter

- Thread common instructions
- Decode/dispatch uncommon instructions
Can we predecode?

**Challenges**

- Intermediate presentation can be as complicated as the original instruction template.
- Code discovery: How we accurately find instruction boundaries?

**Possible Solutions**

- Use small set of intermediate presentation templates.
- Predecode a single CISC into multiple simple predecoded instructions.
- Two-stage process: decode-and-dispatch followed by predecode
Binary Translation
Binary Translation

• What is it? converting the source binary program into a target binary program.

• Source binary instruction $\rightarrow$ customized target code
  - Get rid of instruction “parsing” and jumps altogether
**Binary Translation Example**

**x86 Source Binary**

```
addl %edx, 4(%eax)
movl 4(%eax), %edx
add %eax, 4
```

**Translate to PowerPC Target**

```
r1 points to x86 register context block
r2 points to x86 memory image
r3 contains x86 ISA PC value
```

```
lwz r4, 0(r1) ;load %eax from register block
addi r5, r4, 4 ;add 4 to %eax
lwzx r5, r2, r5 ;load operand from memory
lwz r4, 12(r1) ;load %edx from register block
add r5, r4, r5 ;perform add
stw r5, 12(r1) ;put result into %edx
addi r3, r3, 3 ;update PC (3 bytes)
lwz r4, 0(r1) ;load %eax from register block
addi r5, r4, 4 ;add 4 to %eax
lwz r4, 12(r1) ;load %edx from register block
stwx r4, r2, r5 ;store %edx value into memory
addi r3, r3, 3 ;update PC (3 bytes)
lwz r4, 0(r1) ;load %eax from register block
addi r4, r4, 4 ;add immediate
stw r4, 0(r1) ;place result back into %eax
addi r3, r3, 3 ;update PC (3 bytes)
```
Binary Translation
Optimization: Register Mapping

- Keep copy of register state in target memory
- Easier if $\#_{\text{target}} > \#_{\text{source}}$
- Register mapping may be on a per-block basis
  - If $\#_{\text{target}}$ registers not enough
- Reduces loads/stores significantly
  - memory access to the context block is eliminated (almost)
r1 points to x86 register context block
r2 points to x86 memory image
r3 contains x86 ISA PC value
r4 holds x86 register %eax
r7 holds x86 register %edx
etc.

```
addi     r16,r4,4        ;add 4 to %eax
lwzx     r17,r2,r16     ;load operand from memory
add       r7,r17,r7      ;perform add of %edx
addi     r16,r4,4        ;add 4 to %eax
stwx     r7,r2,r16      ;store %edx value into memory
addi     r4,r4,4         ;increment %eax
addi     r3,r3,9         ;update PC (9 bytes)
```

x86 Source Binary

```
addl %edx,4(%eax)
movl 4(%eax),%edx
addl %eax,4
```

Issues in Binary Translations

• Why can’t we just predecode the whole source ISA to target ISA before execution? → code discovery problem

• The relationship between the source PC (SPC) and target PC (TPC)
The Code Discovery Problem

• There are a number of contributors to the code discovery problem

source ISA instructions

<table>
<thead>
<tr>
<th>inst. 1</th>
<th>inst. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>jump</td>
</tr>
<tr>
<td>inst. 3</td>
<td></td>
</tr>
<tr>
<td>reg.</td>
<td>data</td>
</tr>
<tr>
<td>inst. 5</td>
<td>inst. 6</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>uncond. brnch</td>
<td>pad</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>inst. 8</td>
<td></td>
</tr>
</tbody>
</table>

data in instruction stream

jump indirect to???

pad for instruction alignment
The Code Location Problem

- Source and target binaries use different PCs
- On indirect jump, PC is source, but jump should go to target
Simplified Special Cases

- **Fixed Mapping**
  - All instructions fixed length on known boundaries
  - Like many RISCs

- **Designed Such that:**
  - No jumps or branches to arbitrary locations
    - All jumps via procedure (method) call
  - No data or pads mixed with instructions
  - Example: Java bytecode has these properties
    - All code in a method can be “discovered” when method is entered
Dynamic Translation

• First Interpret
  – And perform code discovery

• Translate Code
  – Incrementally, as it is discovered
  – Place translated blocks into Code Memory
  – Save source to target PC mappings in lookup table

• Emulation process
  – Execute translated block to end
  – Lookup next source PC in table
    • If translated, jump to target PC
    • Else interpret and translate
Dynamic Translation

Source binary code

Translated code

SPC to TPC Lookup Table

Emulation Manager

Interpreter

Translator

Source binary code

data access  flow of control
Dynamic Translation

What is the unit of translation?
Basic Blocks

• Definition: maximal sequence of consecutive instructions such that
  – Flow of control can only enter the basic block from the first instruction
  – Control leaves the block only at the last instruction

• Each instruction is assigned to exactly one basic block
1) $i = 1$
2) $j = 1$
3) $t1 = 10 * i$
4) $t2 = t1 + j$
5) $t3 = 8 * t2$
6) $t4 = t3 - 88$
7) $a[t4] = 0.0$
8) $j = j + 1$
9) if $j <= 10$ goto (3)
10) $i = i + 1$
11) if $i <= 10$ goto (2)
12) $i = 1$
13) $t5 = i - 1$
14) $t6 = 88 * t5$
15) $a[t6] = 1.0$
16) $i = i + 1$
17) if $i <= 10$ goto (13)
Fist we determine *leader* instructions:

1. The first three-address instruction in the intermediate code is a leader.

2. Any instruction that is the target of a conditional or unconditional jump is a leader.

3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

```
1)  i = 1  
2)  j = 1  
3)  t1 = 10 * i  
4)  t2 = t1 + j  
5)  t3 = 8 * t2  
6)  t4 = t3 - 88  
7)  a[t4] = 0.0  
8)  j = j + 1  
9)  if j <= 10 goto (3)  
10) i = i + 1  
11) if i <= 10 goto (2)  
12) i = 1  
13) t5 = i - 1  
14) t6 = 88 * t5  
15) a[t6] = 1.0  
16) i = i + 1  
17) if i <= 10 goto (13)  
```
First we determine *leader* instructions:

1. The first three-address instruction in the intermediate code is a leader.
   1) \( i = 1 \)
   2) \( j = 1 \)
   
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
   3) \( t1 = 10 \* i \)
   4) \( t2 = t1 + j \)
   5) \( t3 = 8 \* t2 \)
   6) \( t4 = t3 - 88 \)
   7) \( a[t4] = 0.0 \)
   8) \( j = j + 1 \)
   9) if \( j \leq 10 \) goto (3)

3. Any instruction that immediately follows a conditional or unconditional jump is a leader.
   10) \( i = i + 1 \)
   11) if \( i \leq 10 \) goto (2)
   12) \( i = 1 \)
   13) \( t5 = i - 1 \)
   14) \( t6 = 88 \* t5 \)
   15) \( a[t6] = 1.0 \)
   16) \( i = i + 1 \)
   17) if \( i \leq 10 \) goto (13)
First we determine *leader* instructions:

1. The first three-address instruction in the intermediate code is a leader.
   
   $i = 1$
   
   $j = 1$

2. Any instruction that is the target of a conditional or unconditional jump is a leader.

   $t1 = 10 \times i$
   
   $t2 = t1 + j$
   
   $t3 = 8 \times t2$
   
   $t4 = t3 - 88$
   
   $a[t4] = 0.0$
   
   $j = j + 1$
   
   if $j \leq 10$ goto (3)
   
   $i = i + 1$
   
   if $i \leq 10$ goto (2)
   
   $i = 1$

3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

   $t5 = i - 1$
   
   $t6 = 88 \times t5$
   
   $a[t6] = 1.0$
   
   $i = i + 1$
   
   if $i \leq 10$ goto (13)

Basic block starts with a leader instruction and stops before the following leader instruction.
These are called Static basic blocks!

But

What we are interested in are Dynamic basic blocks
Dynamic Blocks

<table>
<thead>
<tr>
<th>Static Basic Blocks</th>
<th>Dynamic Basic Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>add...</td>
<td>add...</td>
</tr>
<tr>
<td>load...</td>
<td>load...</td>
</tr>
<tr>
<td>store...</td>
<td>store...</td>
</tr>
<tr>
<td>loop:</td>
<td>loop:</td>
</tr>
<tr>
<td></td>
<td>load ...</td>
</tr>
<tr>
<td></td>
<td>add .....</td>
</tr>
<tr>
<td></td>
<td>store</td>
</tr>
<tr>
<td></td>
<td>brcond skip</td>
</tr>
<tr>
<td></td>
<td>load...</td>
</tr>
<tr>
<td></td>
<td>sub...</td>
</tr>
<tr>
<td>skip:</td>
<td>skip:</td>
</tr>
<tr>
<td></td>
<td>add...</td>
</tr>
<tr>
<td></td>
<td>store</td>
</tr>
<tr>
<td></td>
<td>brcond loop</td>
</tr>
<tr>
<td></td>
<td>add...</td>
</tr>
<tr>
<td></td>
<td>load...</td>
</tr>
<tr>
<td></td>
<td>store...</td>
</tr>
<tr>
<td></td>
<td>jump indirect</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

- A dynamic block always begins at the instruction following a branch/jump.
- A static instruction can belong to several dynamic blocks
Flow of Control

- Control flows between translated blocks and Emulation Manager \( \rightarrow \) an overhead! \( \rightarrow \) Can we optimize that?
Optimization: Translation Chaining

- Jump from one translation directly to next
  - Avoid switching back to EM

**Without Chaining:**

**With Chaining:**
Optimization: Translation Chaining

- Translation-terminating branch is
  - Unconditional: jump directly to next translation
  - Conditional: link each exit separately
- Indirect jumps:
  - Simple Solution
    - Always return to EM
  - Dynamic Chaining
    - Software jump prediction
Optimization: Software Jump Prediction

• Example Code:
  Say Rx holds source branch address
  – addr_i are predicted addresses (in probability order)
    • Determined via profiling
  – target_i are corresponding target code blocks

If Rx == addr_1 goto target_1
Else if Rx == addr_2 goto target_2
Else if Rx == addr_3 goto target_3
Else hash_lookup(Rx) ; do it the slow way
Source/Target ISA Issues

• Register architectures
  – Number of registers in source vs target ISAs
  – We need at least to use the following registers from target ISA (can consume between 3 to 10 target registers):
    • point to register control block
    • point to source memory
    • assigned TPC
    • If interpretation is used the another register to hold SPC
    • one for stack register
    • one for condition codes
    • some frequently special registers
  – The above list does not include general purpose registers from source too!
Source/Target ISA Issues

• Condition codes
• Data formats and operations
  – Floating point
  – Decimal
  – ...
• Address resolution
  – Byte vs Word addressing
• Address alignment
  – Natural vs arbitrary
• Byte order
  – Big/Little endian
Same ISA Emulation: Do we really need it?

- Simulation where dynamic characteristics are collected
- Same ISA but different guest and host OS
- Discovering and managing certain privileged instructions in some system VMs
- For security
- For optimization
Let's Summarize
Emulation Summary

- **Decode/Dispatch Interpretation**
  - Memory Requirements: Low
  - Startup: Fast
  - Steady State Perf.: Slow
  - Portability: Excellent
Emulation Summary

• Indirect Threaded Interpretation
  – Memory Requirements: Low
    • Slightly more than decode/dispatch due to replicated dispatch code
  – Startup: Fast
  – Steady State Perf.: Slow
  – Portability: Excellent
Emulation Summary

• Direct Threaded Interpretation
  – Memory Requirements: High
  – Startup: Slow
  – Steady State Perf.: Medium
  – Portability: Medium
    • Predecoded version is location-specific
Emulation Summary

- **Binary Translation**
  - Memory Requirements: High
  - Startup: Very Slow
  - Steady State Perf.: Fast
  - Portability: Poor
Conclusions

• In this lecture we looked at emulation. A technique used in most VMs.

• Two types:
  – Interpretation
    • Decode & Dispatch
    • Threaded interpretation
    • Predecoding
    • Direct thread interpretation
  – Dynamic translation
    • Code discovery
    • Code location (SPC & TPC)
    • Optimization: chaining
    • Optimization: indirect jump prediction