



CSCI-GA.3033-015

Virtual Machines: Concepts & Applications

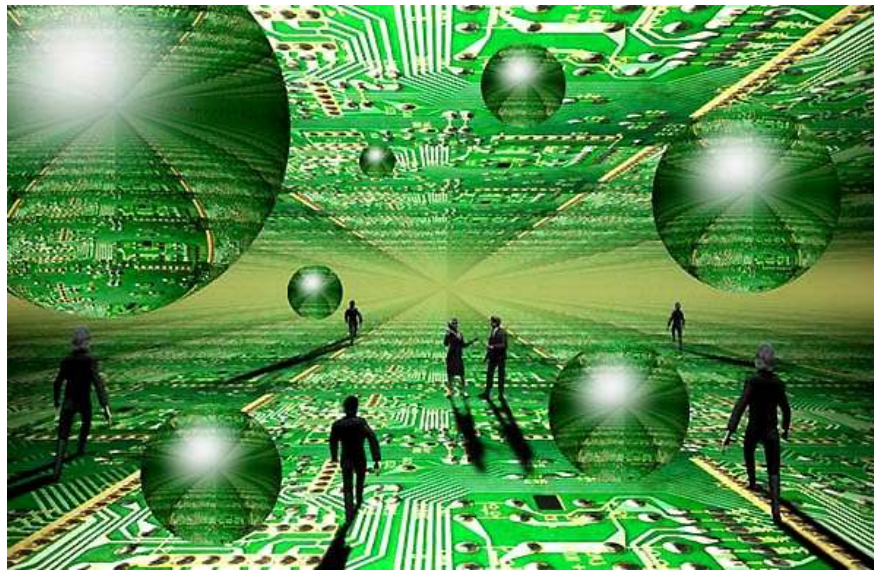
Lecture 2: The Art of Emulation

Mohamed Zahran (aka Z)

mzahran@cs.nyu.edu

<http://www.mzahran.com>

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

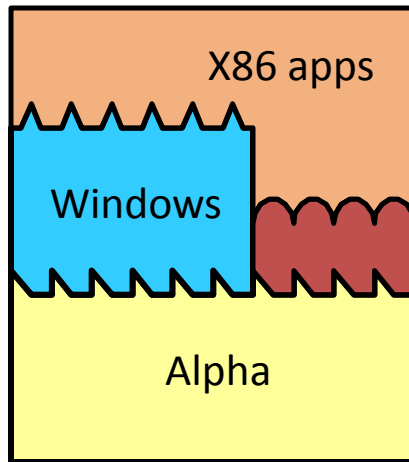


Emulation

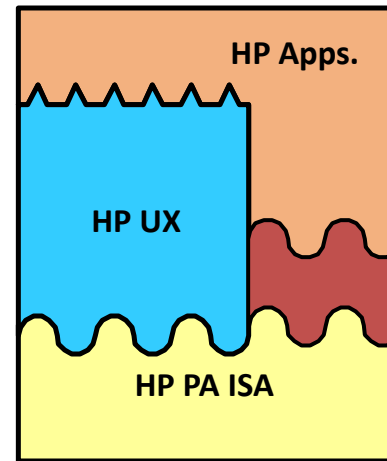
- **Why** are we studying it here?
 - Techniques that can be applied to any type of VM.
- **Definition:** The process of implementation the interface and functionality of one system on a system with different interface and functionality.
- **Science:** How to do it?
- **Art:** How to do it well? → performance

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)



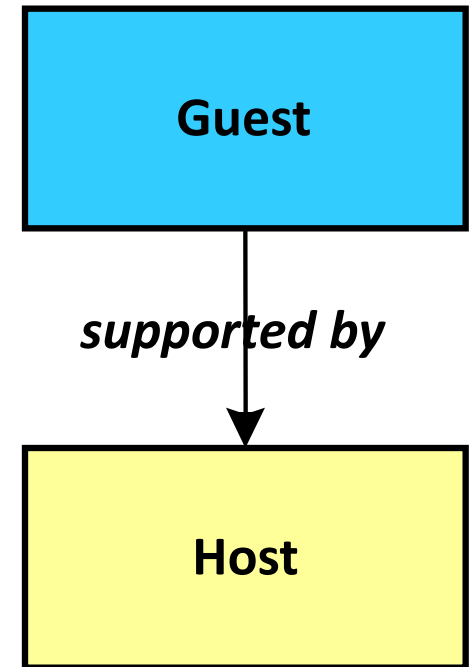
Emulation



Optimization

Definitions

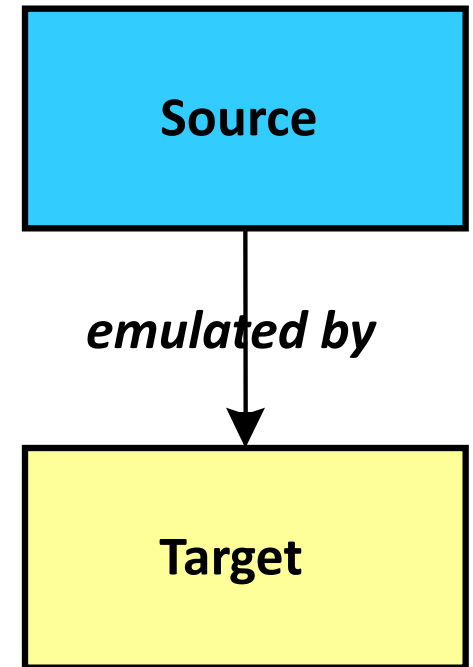
- *Guest*
 - Environment that is being supported by underlying platform
- *Host*
 - Underlying platform that provides guest environment



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.

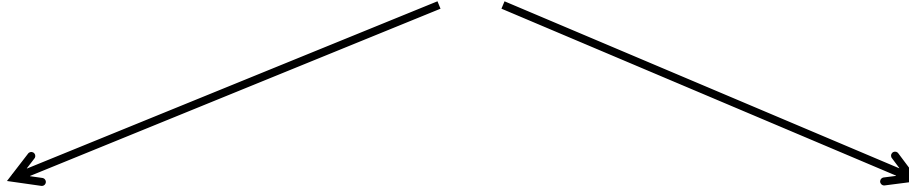


Warning: You may not find standard definition in literature ... Unfortunately!

How to do emulation?



```
graph TD; A[How to do emulation?] --> B[Interpretation]; A --> C[Binary Translation];
```



Interpretation

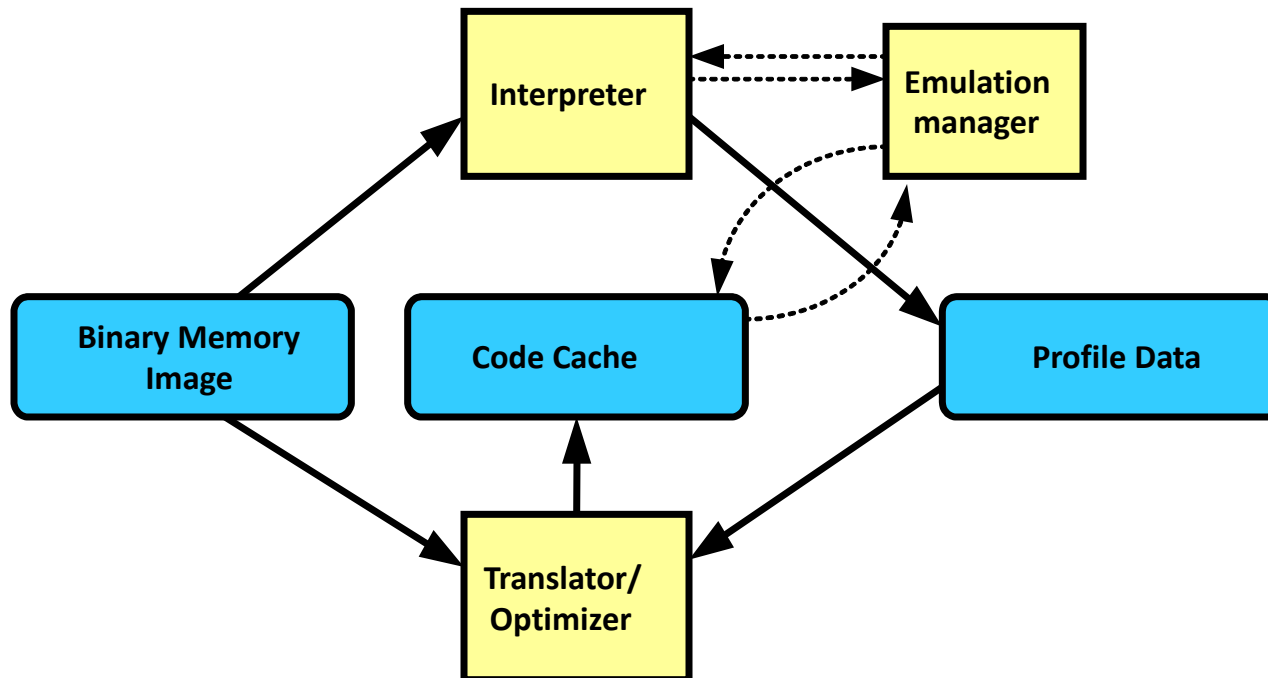
- Fetch source instruction
- Analyze it
- Perform the required operation

Binary Translation

- block of source instructions
→ block of target instructions
- Save for repeated use

Big Picture

- ❑ In practice: staged operation
 - Interpretation
 - Binary Translation and Caching
 - Use profiling to determine which

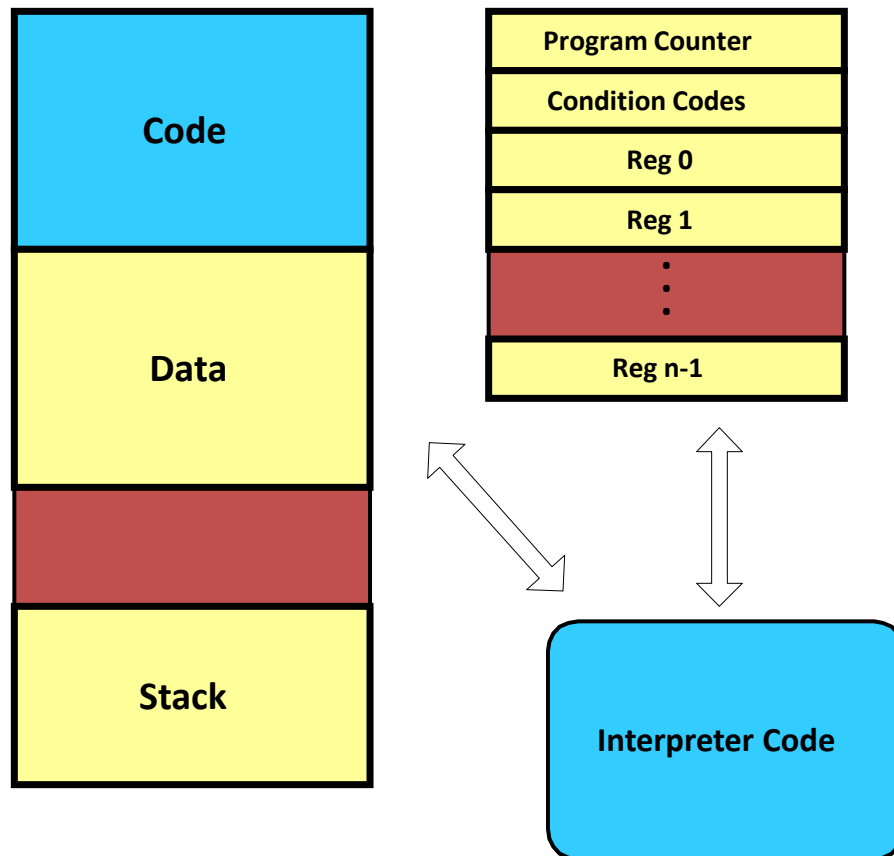


Interpreters

- 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



Version 1: Decode-Dispatch Interpretation

```
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];
    extended_opcode =
        extract(inst,10,10);
    switch(extended_opcode) {
        case Add: Add(inst);
        case AddCarrying:
            AddCarrying(inst);
        case AddExtended:
            AddExtended(inst);
        . . . }
    PC = PC + 4;
}
```

Version 1: Decode-Dispatch Interpretation

```
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];  
    extended_opcode =  
        extract(inst, 10, 10);  
    switch(extended_opcode) {  
        case Add: Add(inst);  
        case AddCarrying:  
            AddCarrying(inst);  
        case AddExtended:  
            AddExtended(inst);  
        . . . }  
    PC = PC + 4;  
}
```

Decode-Dispatch: Efficiency

- Decode-Dispatch Loop
 - Mostly serial code
 - Case statement (hard-to-predict indirect jump)
 - Call to function routine
 - Return
- Executing an add instruction
 - Approximately 20 target instructions
 - Several loads/stores
 - Several shift/mask steps
- Hand-coding can lead to better performance
 - Example: DEC/Compaq FX!32

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

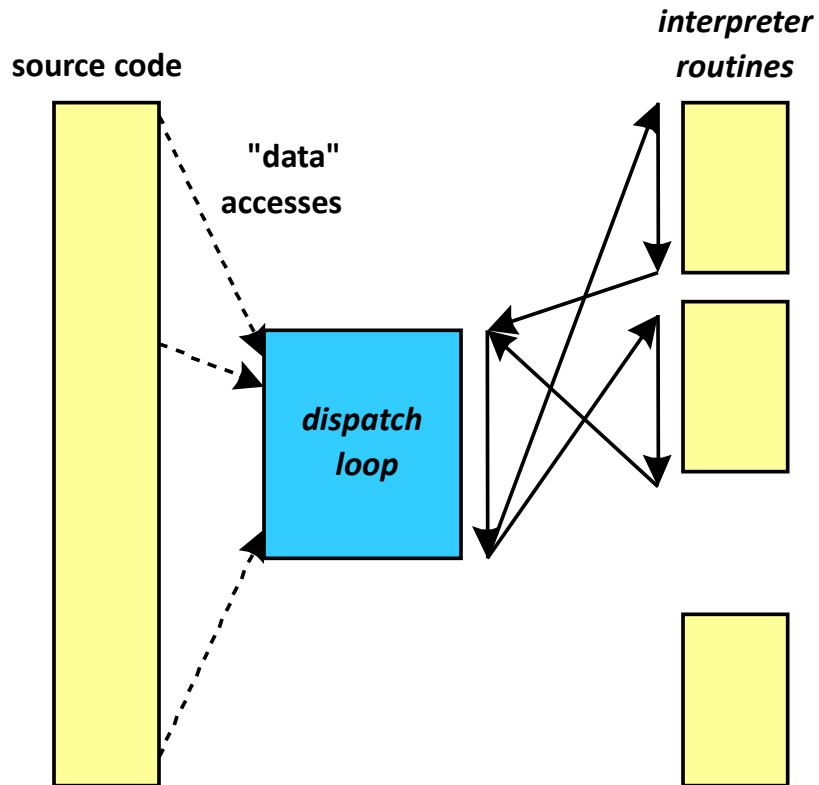
LoadWordAndZero:

```
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;
inst = code[PC];
opcode = extract(inst,31,6)
extended_opcode =
    extract(inst,10,10);
routine =
    dispatch[opcode,extended_opcode];
goto *routine;
```

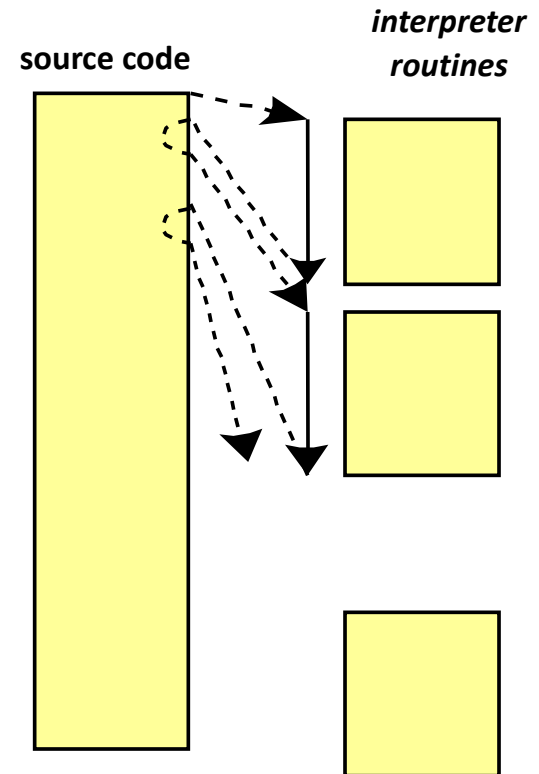
Add:

```
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



Decode-dispatch



Threaded

Version 3: Predecoding

- Scan static instructions and convert to internal pre-decoded form
 - Separate the opcode, operands, etc.
 - Perhaps put into intermediate form
 - Stack form
 - RISC-like micro-ops
 - Reduces shift/masks significantly

lwz **r1, 8(r2)**

add **r3, r3, r1**

stw **r3, 0(r4)**

07		
1	2	08

(load word and zero)

08		
3	1	03

(add)

37		
3	4	00

(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;
```

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

001048d0		
1	2	08

(load word and zero)

00104800		
3	1	03

(add)

00104910		
3	4	00

(store word)

Version 4: Direct Threaded Interpretation

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

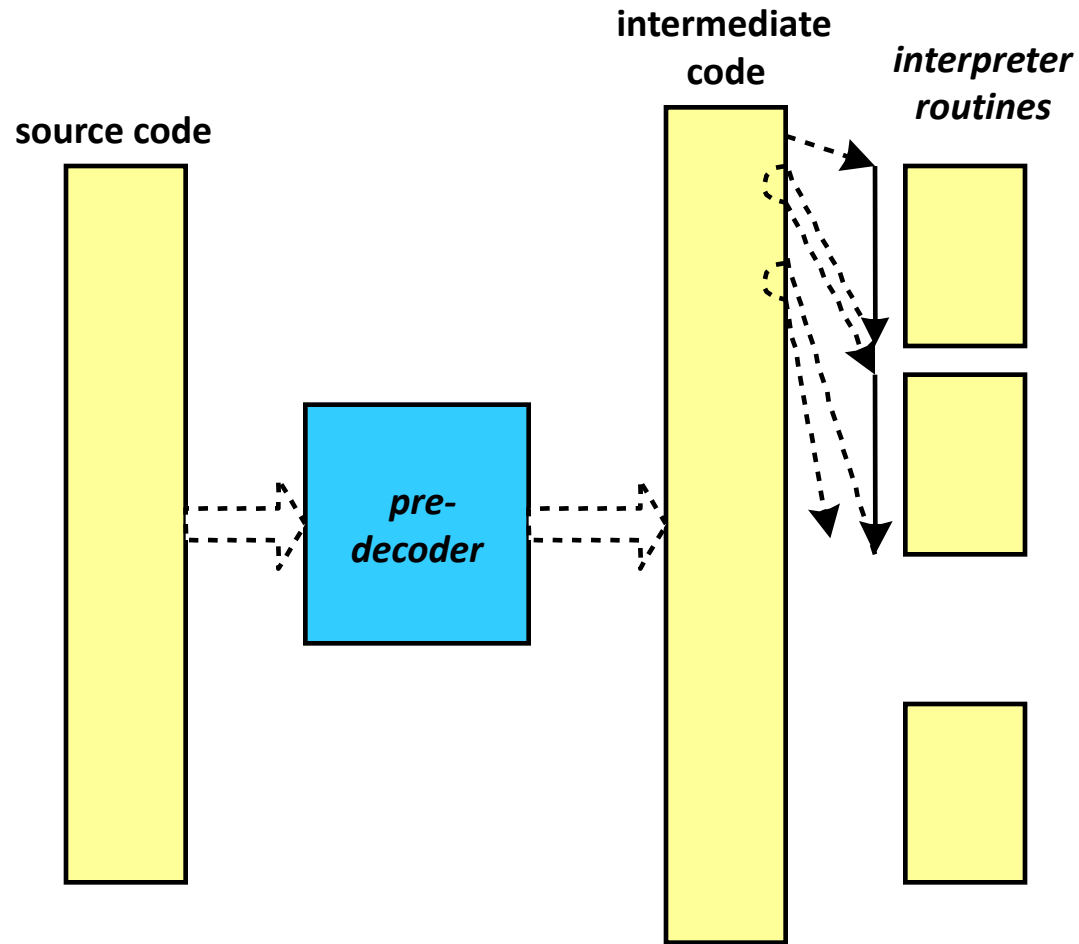
Load Word and Zero:

```
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 = code[TPC].op;  
goto *routine;
```

Version 4: Direct Threaded Interpretation



Binary Translation

- Source binary instruction → 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

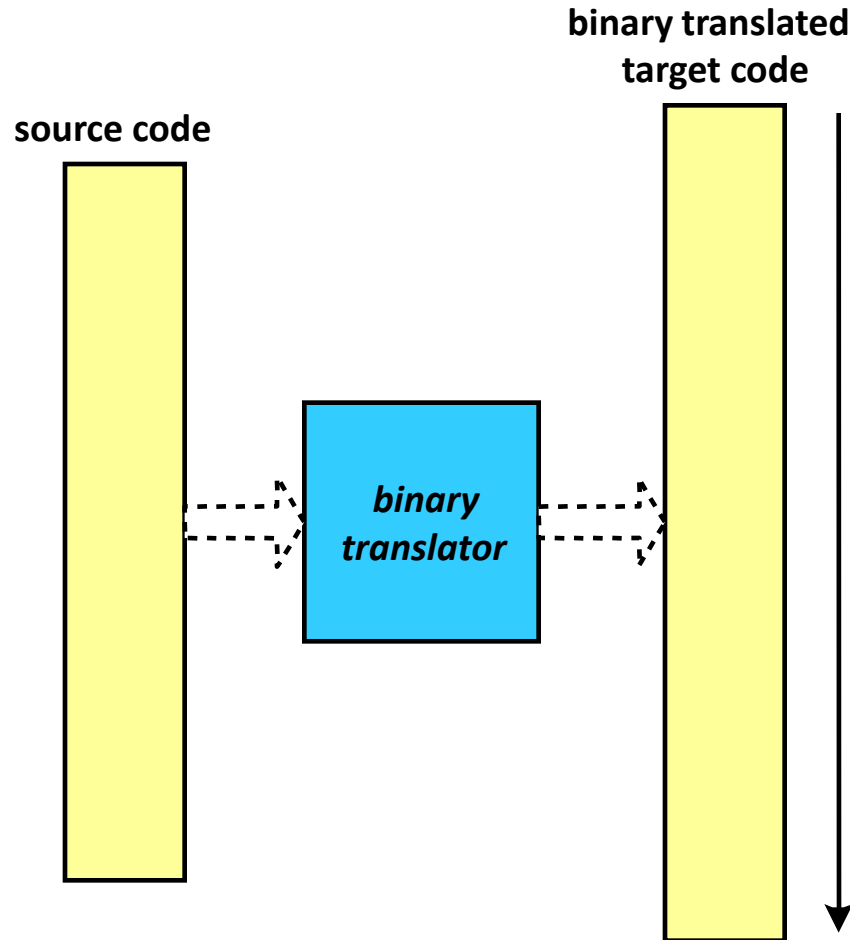
```
r1 points to x86 register context block
r2 points to x86 memory image
r3 contains x86 ISA PC value
```

```
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)
```

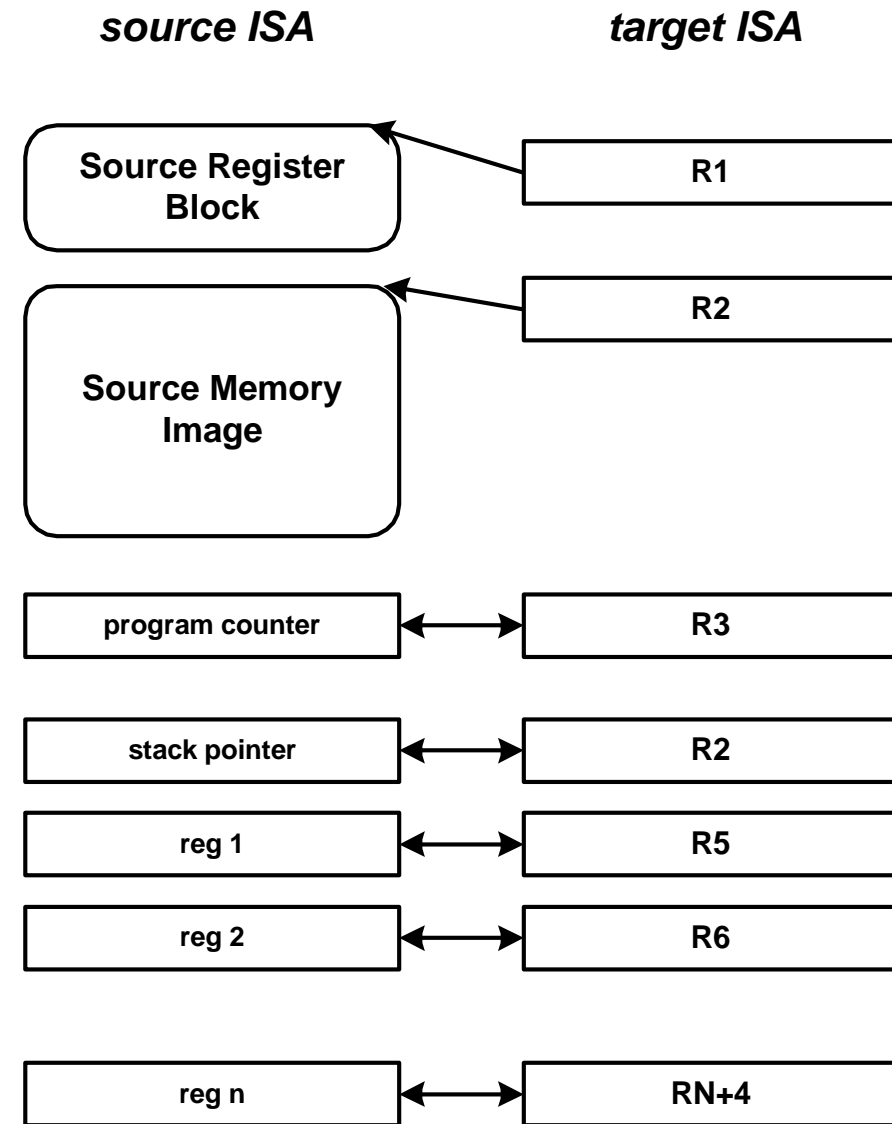
```
lwz      r4, 0(r1)      ;load %eax from register block
addi     r4, r4, 4      ;add immediate
stw      r4, 0(r1)     ;place result back into %eax
addi     r3, r3, 3      ;update PC (3 bytes)
```

Binary Translation



Optimization: Register Mapping

- Keep copy of register state in target memory
 - Spill registers when needed by emulator
- Easier if $\#target\ regs > \#source\ regs$
- Register mapping may be on a per-block basis
 - If $\#target\ registers$ not enough
- Reduces loads/stores significantly

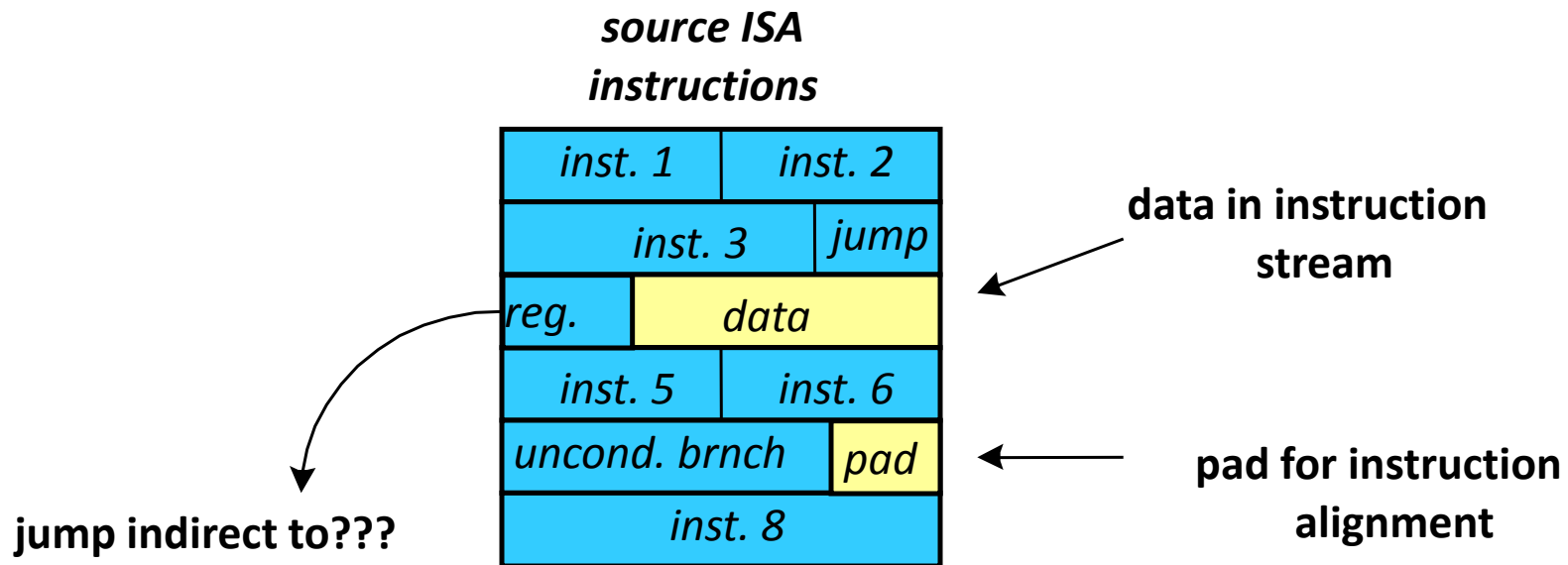


Issues in Binary Translations

- 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



The Code Location Problem

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

x86 source code

```
movl    %eax, 4(%esp)    ;load jump address from memory
jmp     %eax             ;jump indirect through %eax
```

PowerPC target code

```
addi    r16,r11,4        ;compute x86 address
lwzx    r4,r2,r16        ;get x86 jump address
                        ; from x86 memory image
mtctr   r4               ;move to count register
bctr                      ;jump indirect through ctr
```

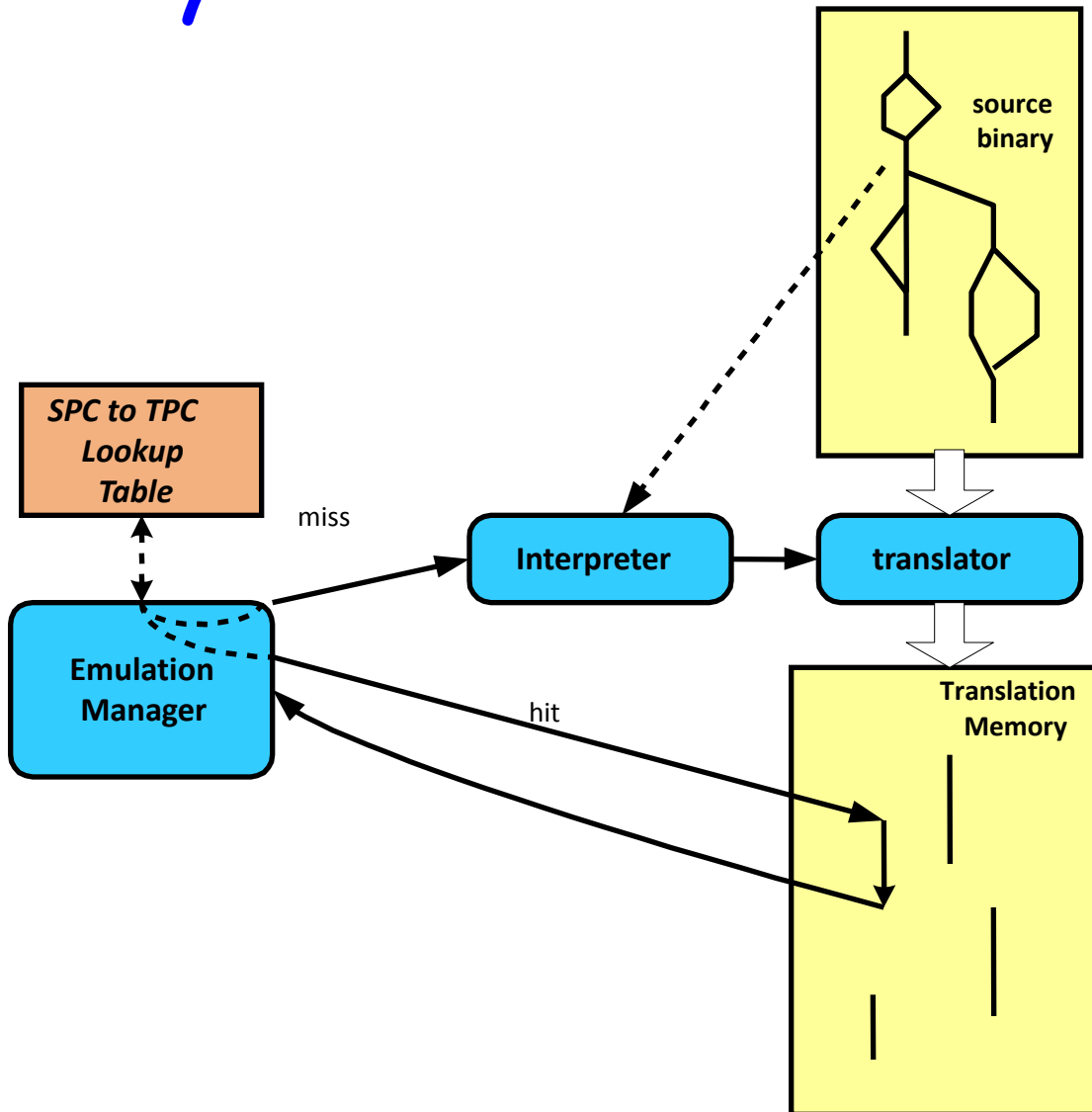
Simplified Special Cases

- Fixed Mapping
 - All instructions fixed length on known boundaries
 - Like many RISCs
- Designed In
 - No jumps or branches to arbitrary locations
 - All jumps via procedure (method) call
 - No data or pads mixed with instructions
 - Java has these properties
 - All code in a method can be “discovered” when method is entered

Dynamic Translation

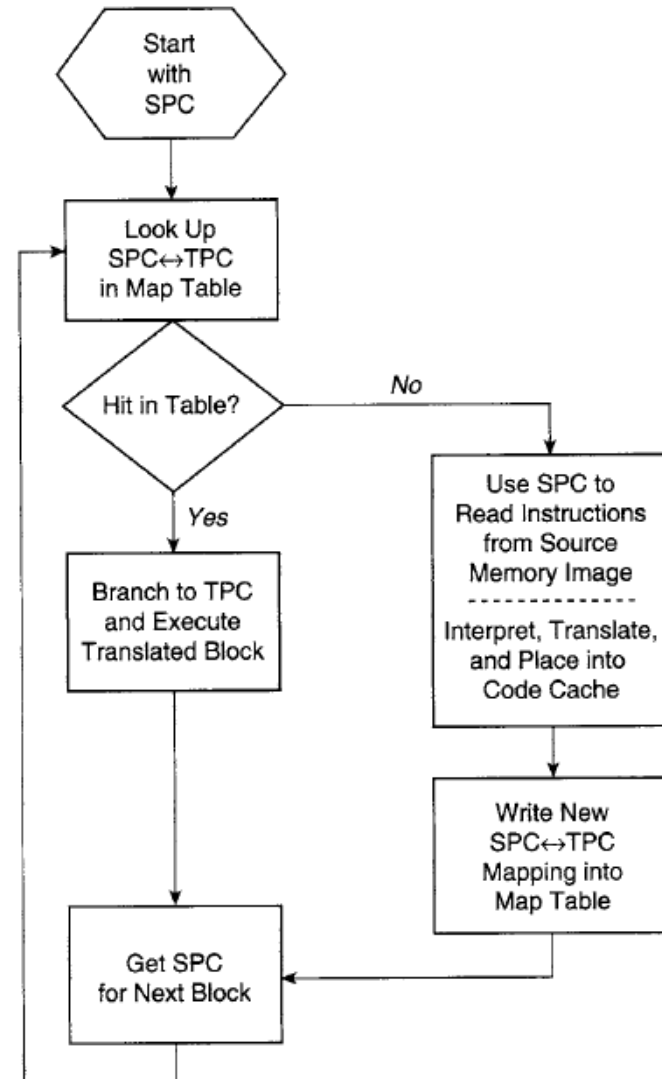
- First Interpret
 - And perform code discovery
- Translate Code
 - Incrementally, as it is discovered
 - Place translated blocks into **Code Memory**
 - Save source to target PC mappings in **lookup table**
- Emulation process
 - Execute translated block to end
 - Lookup next source PC in table
 - If translated, jump to target PC
 - Else interpret and translate

Dynamic Translation



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


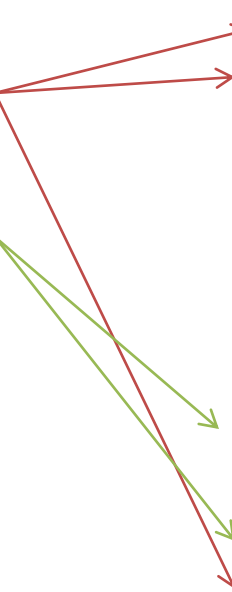
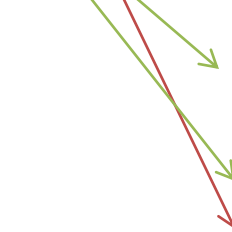

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

Fist we determine *leader* instructions:




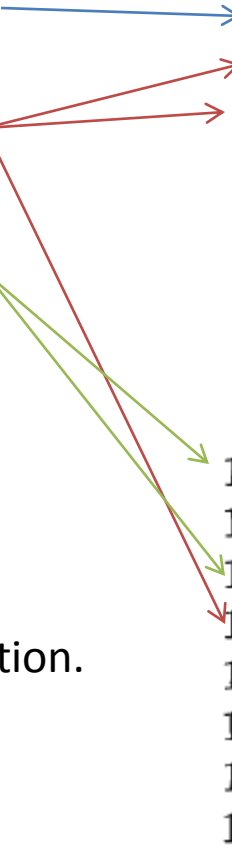
1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

```
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)
```

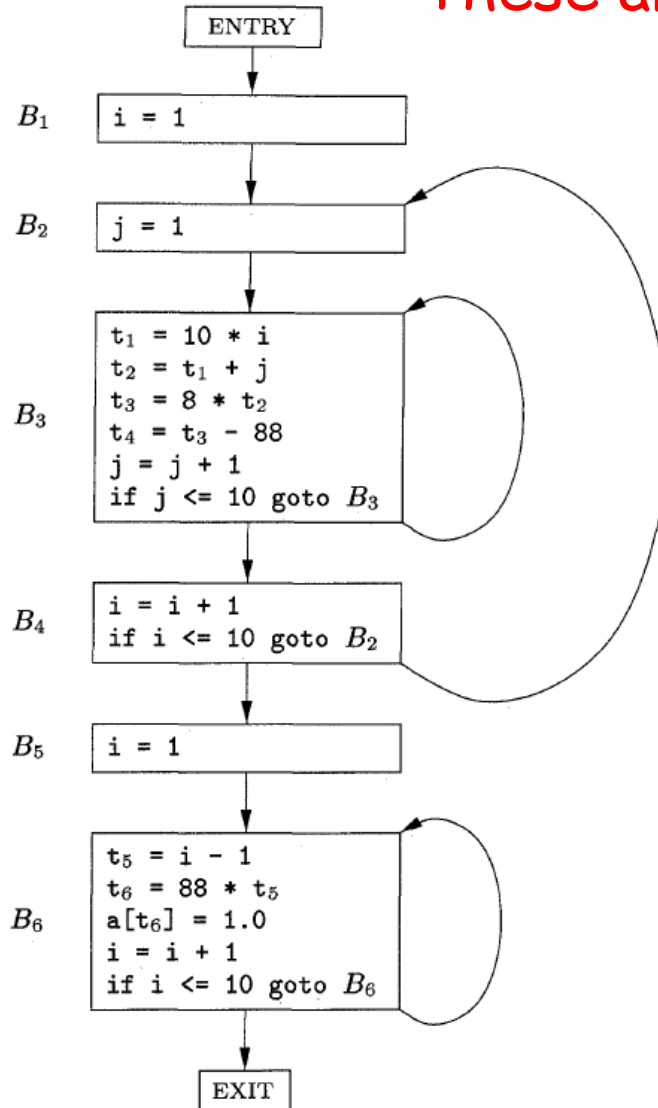
First we determine *leader* instructions:

1. The first three-address instruction in the intermediate code is a leader. 
 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. 
 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. 
- Basic block starts with a leader instruction and stops before the following leader instruction.
- 
- ```
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)
```

These are called Static basic blocks!



But  
What we are interested in are  
**Dynamic basic blocks**

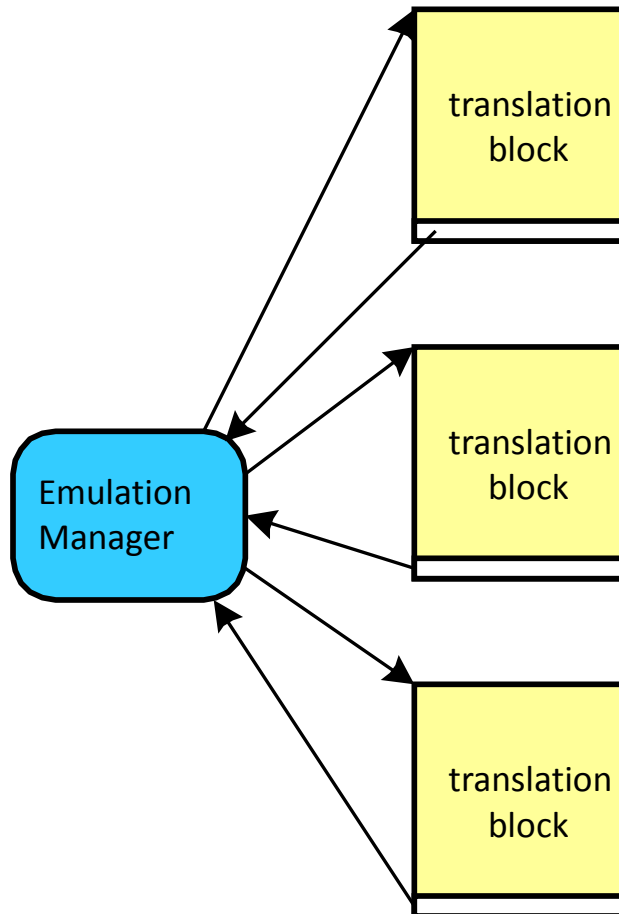
# Dynamic Blocks

| <i>Static<br/>Basic Blocks</i> |                                                         | <i>Dynamic<br/>Basic Blocks</i> |                                                        |
|--------------------------------|---------------------------------------------------------|---------------------------------|--------------------------------------------------------|
|                                | add...<br>load...<br>store ...<br>-----                 |                                 | add...<br>load...<br>store ...<br>-----                |
| loop:                          | load ...<br>add .....<br>store<br>brcond skip<br>-----  | loop:                           | load ...<br>add .....<br>store<br>brcond skip<br>----- |
|                                | load...<br>sub...<br>-----                              |                                 | load...<br>sub...<br>-----                             |
| skip:                          | add...<br>store<br>brcond loop<br>-----                 | skip:                           | add...<br>store<br>brcond loop<br>-----                |
|                                | add...<br>load...<br>store...<br>jump indirect<br>----- | loop:                           | load ...<br>add .....<br>store<br>brcond skip<br>----- |
|                                | ...                                                     | skip:                           | add...<br>store<br>brcond loop<br>-----                |
|                                | ...                                                     |                                 | ...                                                    |
|                                |                                                         |                                 | ...                                                    |

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

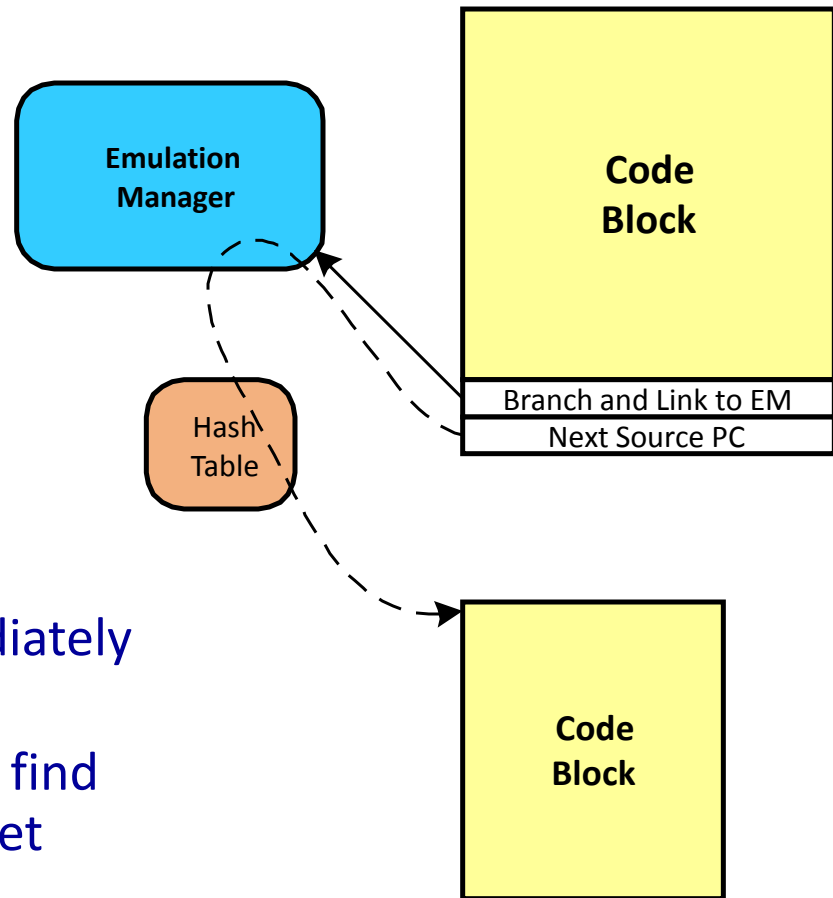
# Flow of Control

- Control flows between translated blocks and Emulation Manager



# Tracking the Source PC

- Can always update SPC as part of translated code
- Better to place SPC in *stub*



## General Method:

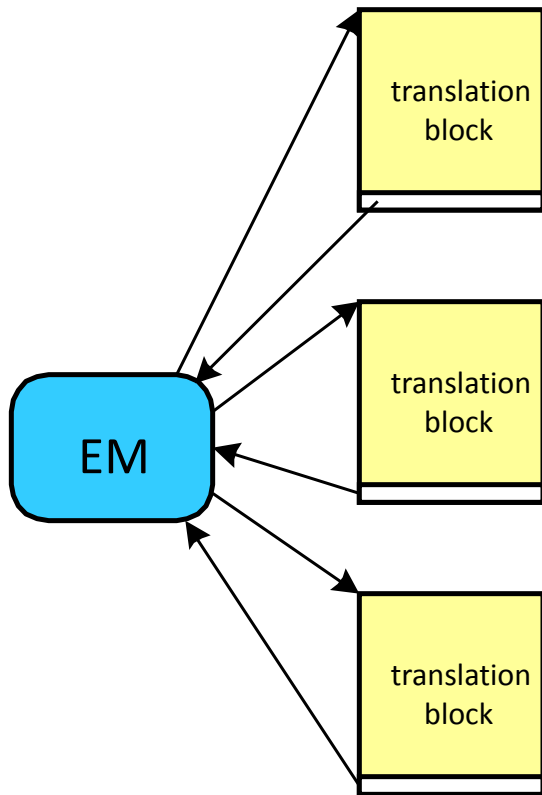
- Translator returns to EM via BL
- Source PC placed in stub immediately after BL
- EM can then use link register to find source PC and hash to next target code block



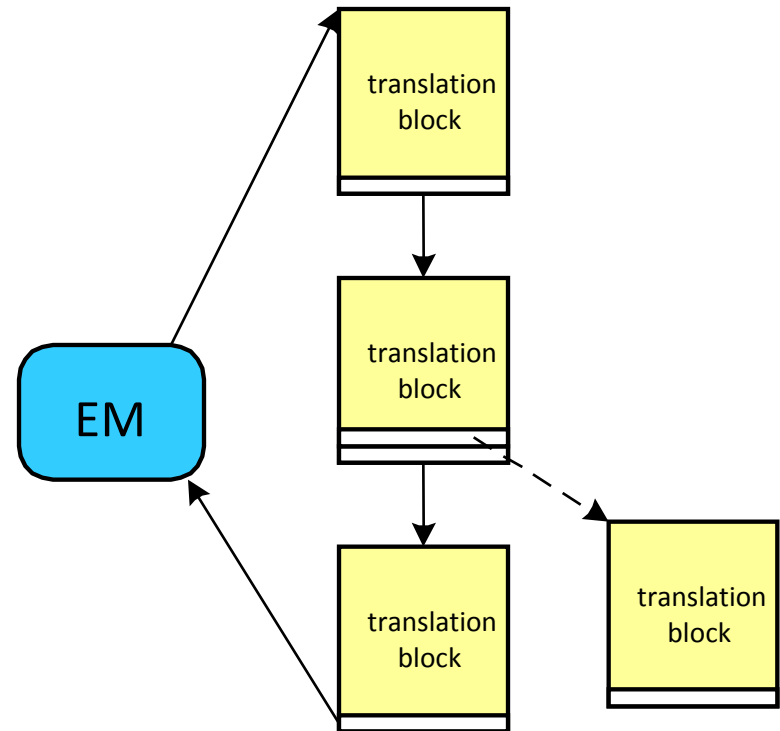
# Optimization: Translation Chaining

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

*Without Chaining:*

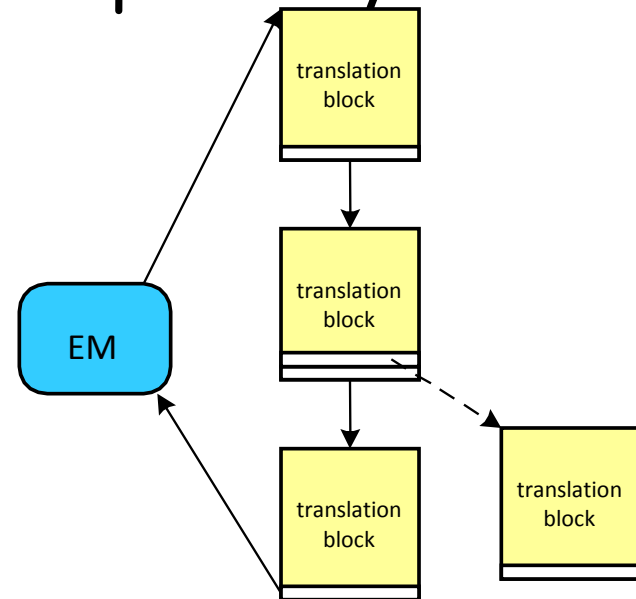


*With Chaining:*



# 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
- Condition codes
- Data formats and operations
  - Floating point
  - Decimal
  - MMX
- 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

# 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