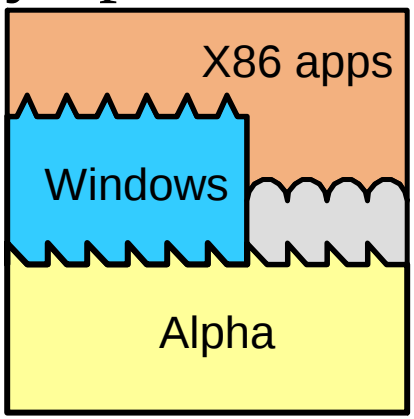
Emulation – Outline

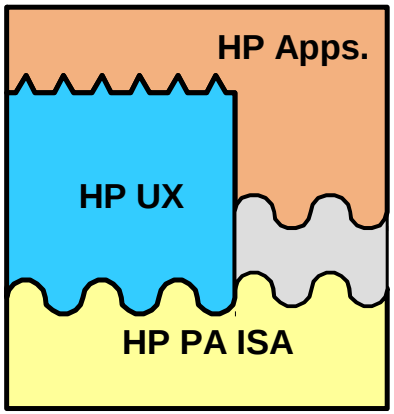
- Emulation
- Interpretation
  - basic, threaded, directed threaded
  - other issues
- Binary translation
  - code discovery, code location
  - other issues
- Control Transfer Optimizations

Key VM Technologies

- Emulation – binary in one ISA is executed in 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



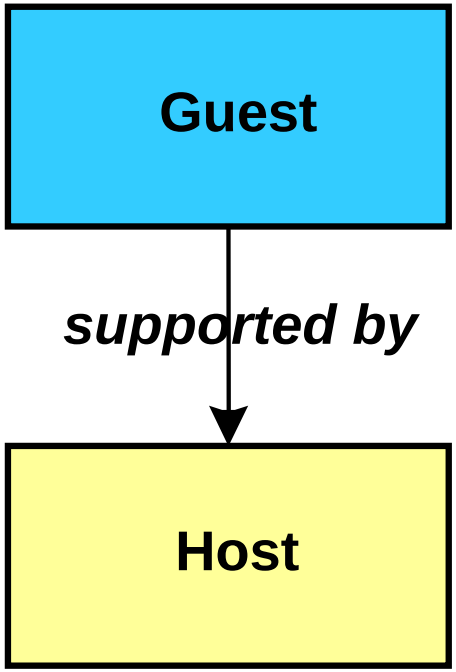
Optimization

Emulation Vs. Simulation

- Emulation
  - method for enabling a (sub)system to present the same interface and characteristics as another
  - ways of implementing emulation
    - *interpretation:* relatively inefficient instruction-at-a-time
    - *binary translation:* block-at-a-time optimized for repeated
  - e.g., the execution of programs compiled for instruction set A on a machine that executes instruction set B.
- Simulation
  - method for modeling a (sub)system's operation
  - objective is to study the process; not just to imitate the function
  - typically emulation is part of the simulation process

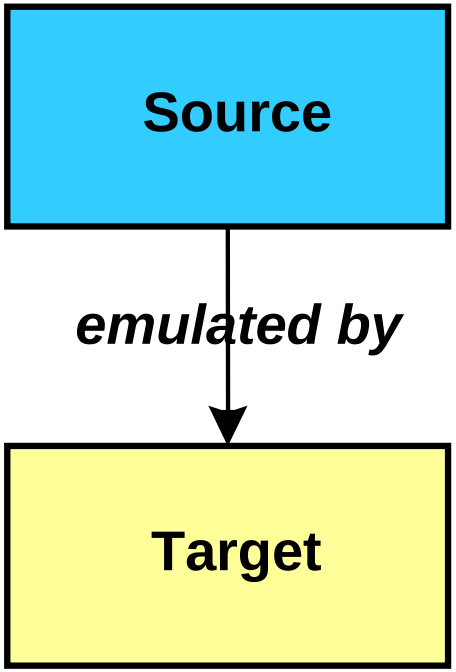
Definitions

- Guest
  - environment being supported by underlying platform
- Host
  - underlying platform that provides guest environment



Definitions (2)

- Source ISA or binary
  - original instruction set or binary
  - the ISA to be emulated
- Target ISA or binary
  - ISA of the host processor
  - underlying ISA
- Source/Target refer to ISAs
- Guest/Host refer to platforms



Emulation

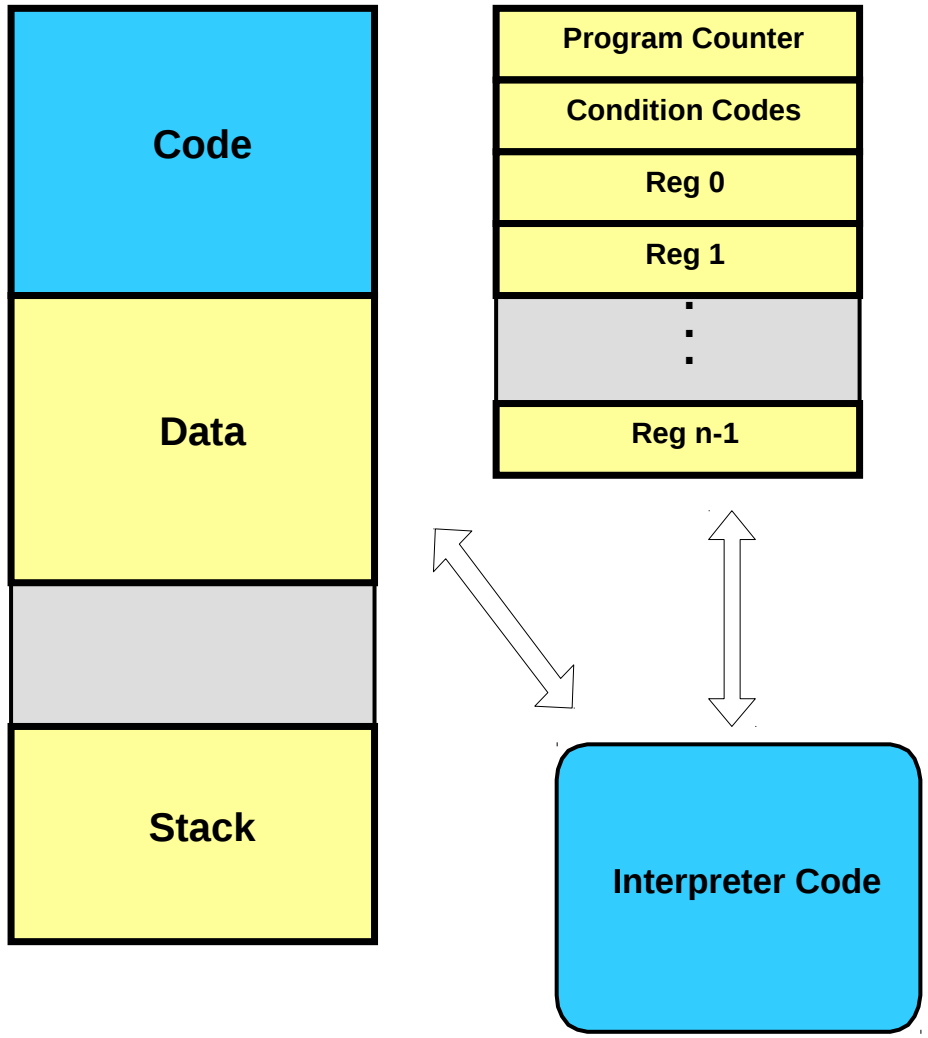
- Required for implementing many VMs.
- Process of implementing the interface and functionality of one (sub)system on a (sub)system having a different interface and functionality
  - terminal emulators, such as for VT100, xterm, putty
- Instruction set emulation
  - binaries in *source* instruction set can be executed on machine implementing *target* instruction set
  - e.g., IA-32 execution layer

Interpretation Vs. Translation

- Interpretation
  - simple and easy to implement, portable
  - low performance
  - threaded interpretation
- Binary translation
  - complex implementation
  - high initial translation cost, small execution cost
  - selective compilation
- We focus on user-level instruction set emulation of program binaries.

Interpreter State

- An interpreter needs to maintain the complete architected state of the machine implementing the source ISA
  - registers
  - memory
    - code
    - data
    - stack



Decode – Dispatch Interpreter

- Decode and dispatch interpreter
  - step through the source program one instruction at a time
  - *decode* the current instruction
  - *dispatch* to corresponding interpreter routine
  - very high interpretation cost

```
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
```

Decode – Dispatch Interpreter (2)

- Instruction function: *Load*

```
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;
}
```

Decode – Dispatch Interpreter (3)

- Instruction function: *ALU*

```
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 and shift/mask steps
- Hand-coding can lead to better performance
  - example: DEC/Compaq FX!32

Indirect Threaded Interpretation

- High number of branches in *decode-dispatch* interpretation reduces performance
  - overhead of 5 branches per instruction
- *Threaded* interpretation improves efficiency by reducing branch overhead
  - append dispatch code with each interpretation routine
  - removes 3 branches
  - *threads* together function routines

Indirect Threaded Interpretation (2)

```
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;
```

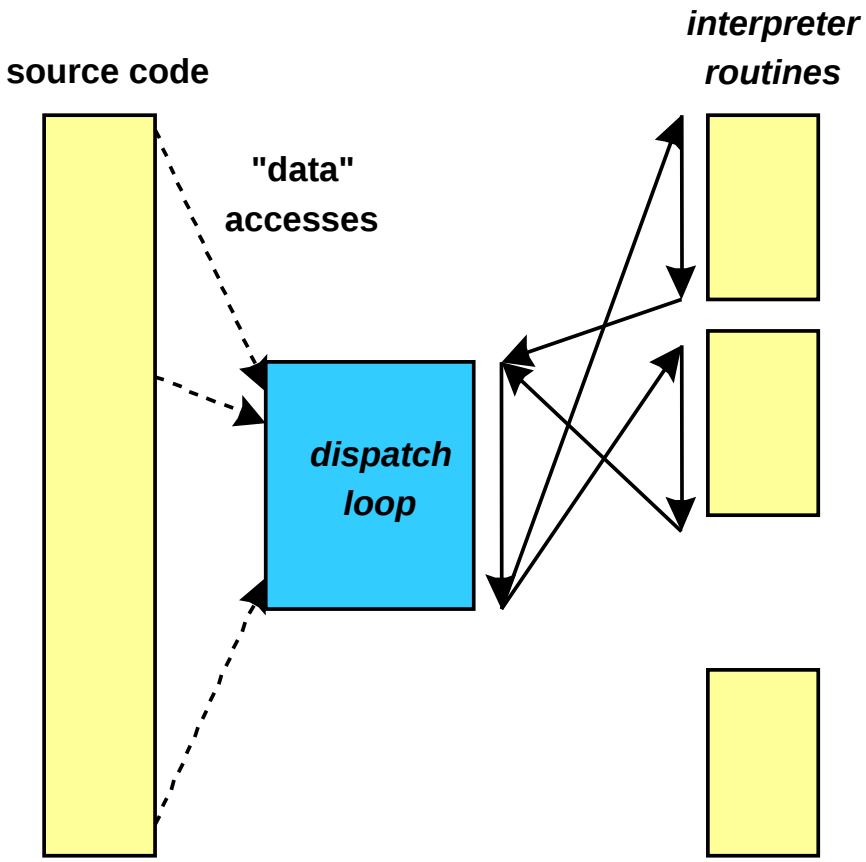
Indirect Threaded Interpretation (3)

```
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;
```

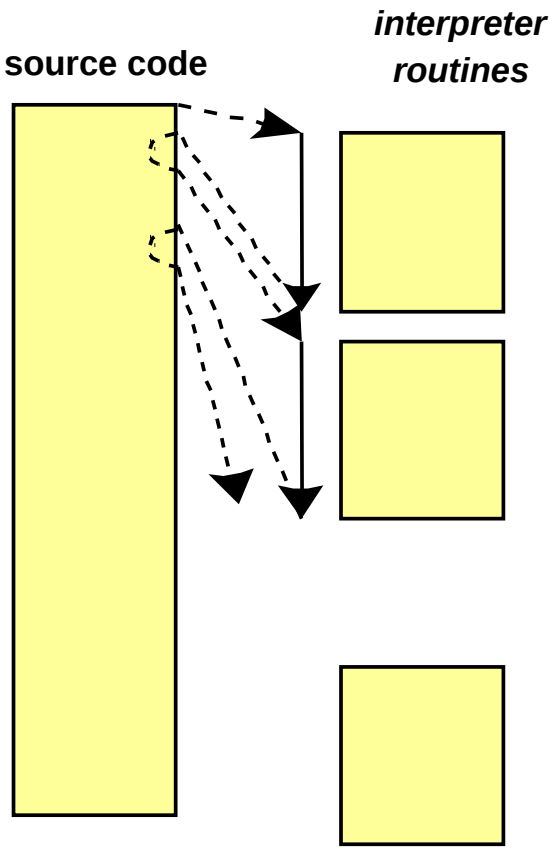
Indirect Threaded Interpretation (4)

- Dispatch occurs *indirectly* through a table
  - interpretation routines can be modified and relocated independently
- Advantages
  - binary intermediate code still portable
  - improves efficiency over basic interpretation
- Disadvantages
  - code replication increases interpreter size

Indirect Threaded Interpretation (5)



Decode-dispatch

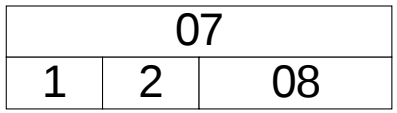


Threaded

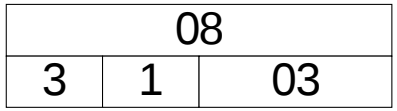
Predecoding

- Parse each instruction into a pre-defined structure to facilitate interpretation
  - separate opcode, operands, etc.
  - reduces shifts / masks significantly
  - more useful for CICS ISAs

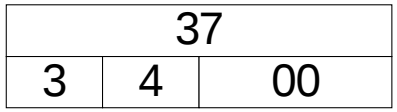
```
lwz     r1, 8(r2)
add     r3, r3,r1
stw     r3, 0(r4)
```



(load word and zero)



(add)



(store word)

Predecoding (2)

```
struct instruction {
   unsigned long op;
   unsigned char dest, src1, 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;
```

Direct Threaded Interpretation

- Allow even higher efficiency by
  - removing the memory access to the centralized table
  - requires predecoding
  - dependent on locations of interpreter routines
    - *loses* portability

001048d0		
1	2	08

(load word and zero)

00104800		
3	1	03

(add)

00104910		
3	4	00

(store word)

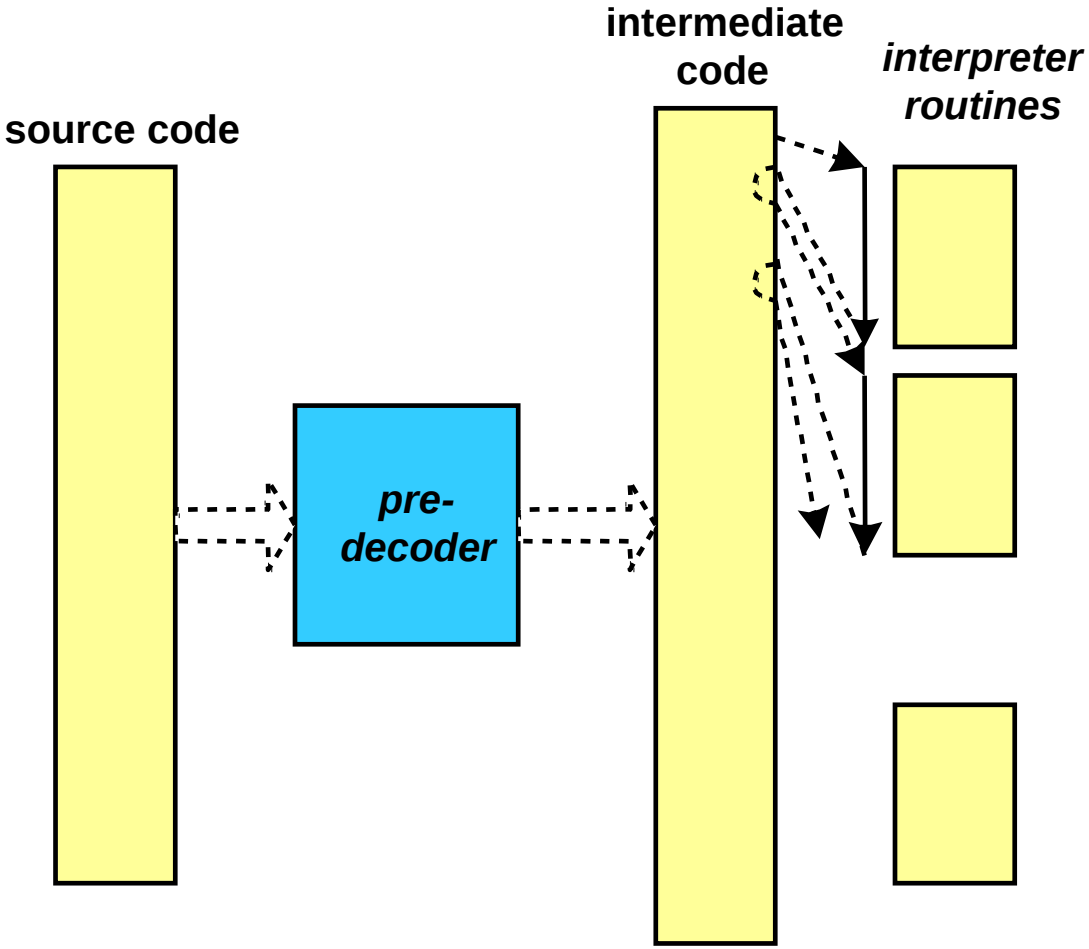
Direct Threaded Interpretation (2)

- Predecode the source binary into an intermediate structure
- Replace the opcode in the intermediate form with the address of the interpreter routine
- Remove the memory lookup of the dispatch table
- Limits portability since exact locations of the interpreter routines are needed

Direct Threaded Interpretation (3)

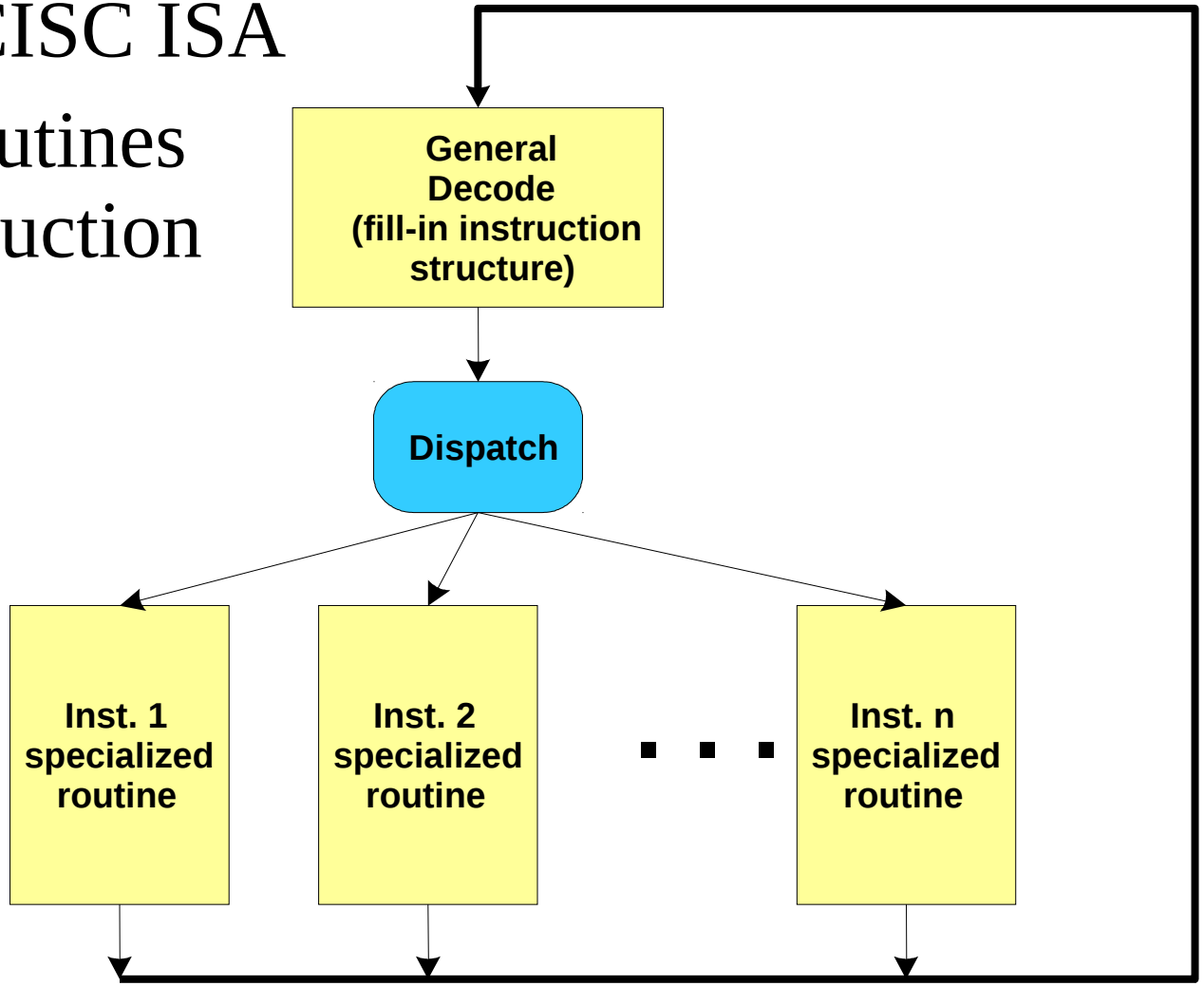
```
Load Word and Zero:
   RT = code[TPC].dest;
   RA = code[TPC].src1;
   displacement = code[TPC].src2;
   if (RA == 0) source = 0;
   else source = regs[RA];
   address = source + displacement;
   regs[RT] = (data[address]<< 32) >> 32;
   SPC = SPC + 4;
   TPC = TPC + 1;
   If (halt || interrupt)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

Direct Threaded Interpretation (4)



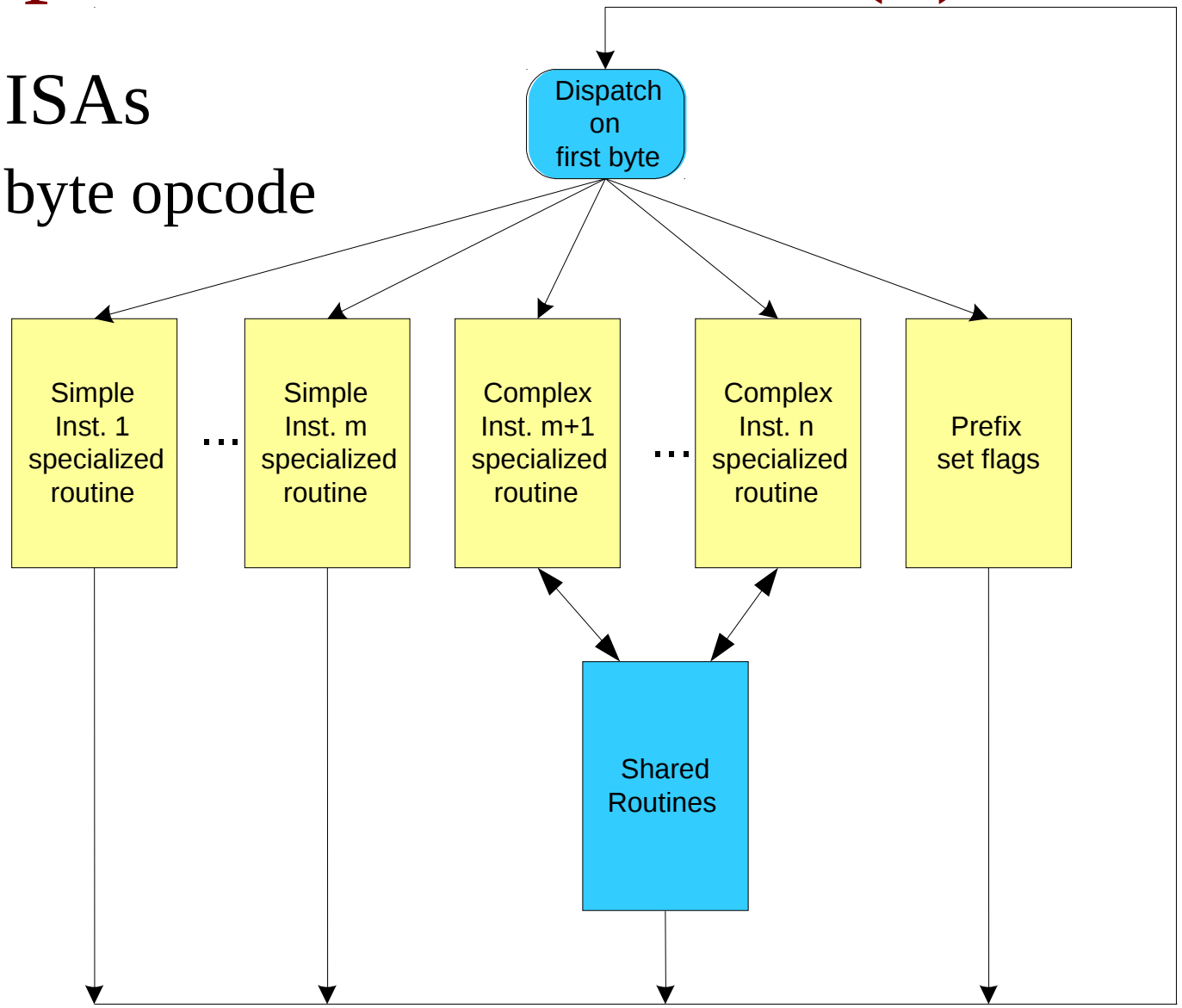
Interpreter Control Flow

- Decode for CISC ISA
- Individual routines for each instruction



Interpreter Control Flow (2)

- For CISC ISAs
  - multiple byte opcode
  - make common cases fast



Binary Translation

- Translate source binary program to target binary before execution
  - is the logical conclusion of predecoding
  - get rid of *parsing* and jumps altogether
  - allows optimizations on the *native* code
  - achieves higher performance than interpretation
  - needs mapping of source state onto the host state (*state mapping)*

Binary Translation (2)

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
```

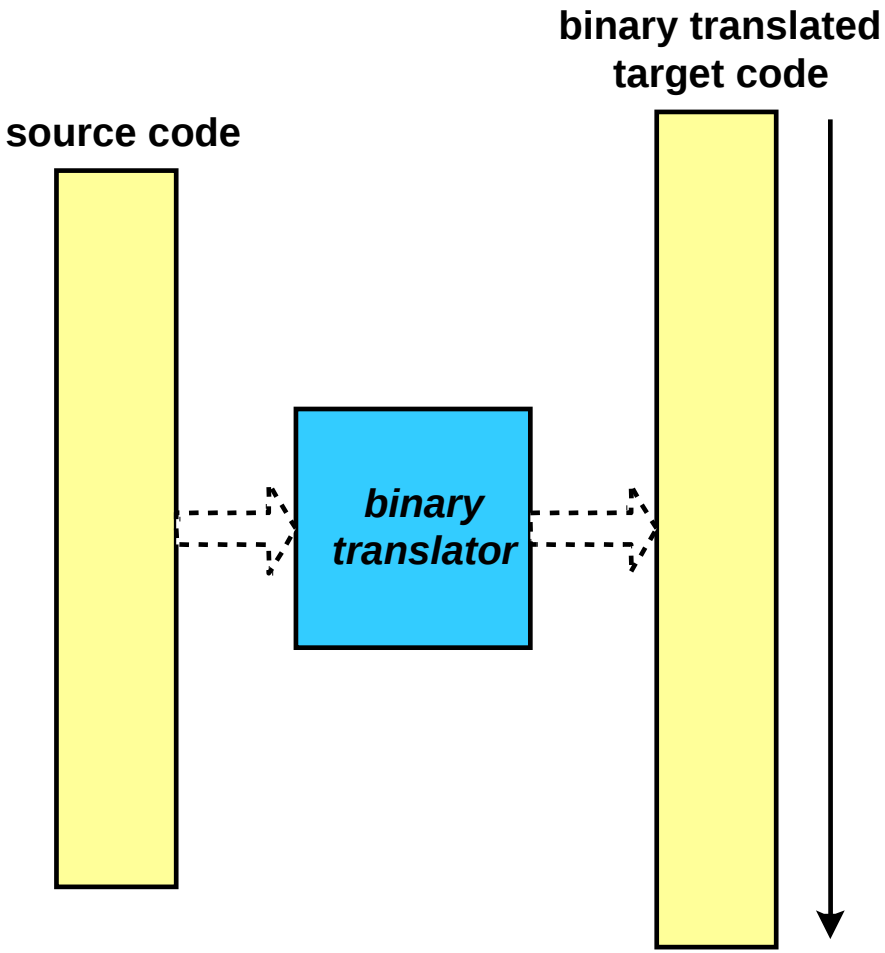
Binary Translation (3)

```
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 (4)

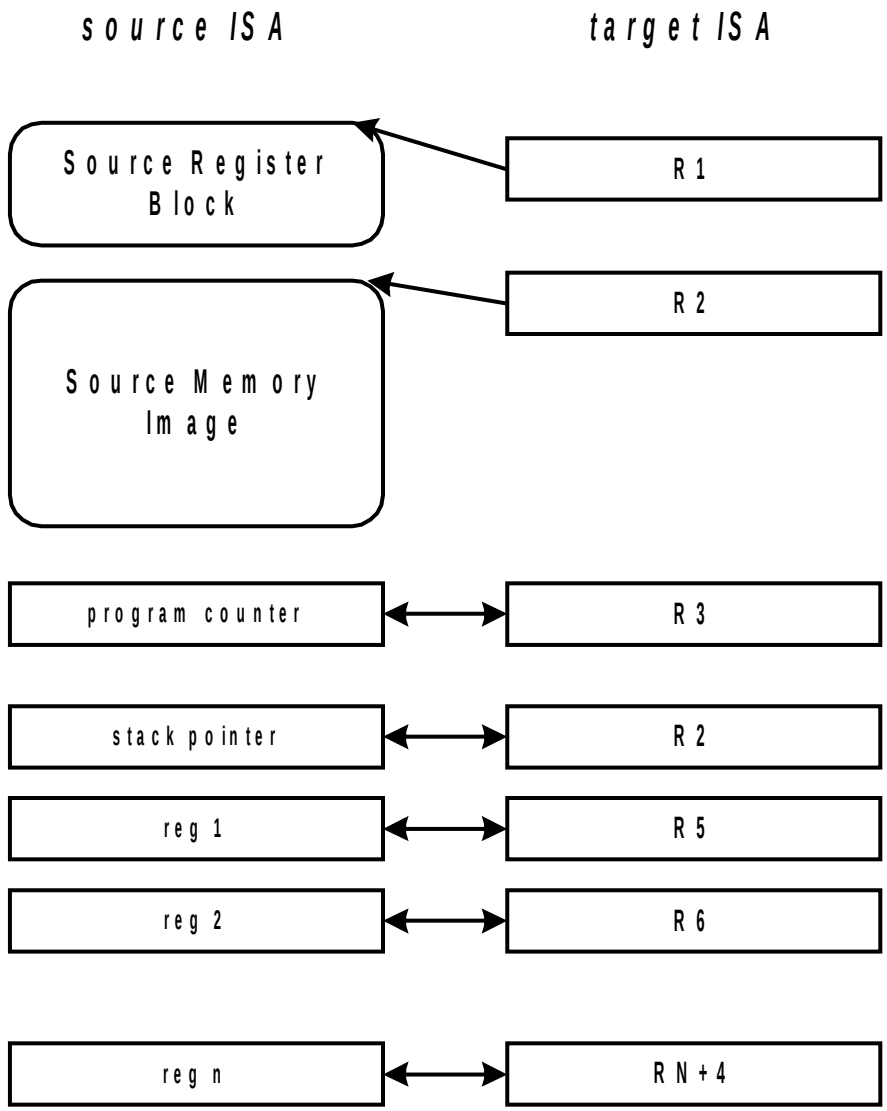


State Mapping

- Maintaining the state of the source machine on the host (target) machine.
  - state includes source registers and memory contents
  - source registers can be held in host registers or in host memory
  - reduces loads/stores significantly
  - easier if target registers > source registers

Register Mapping

- Map source registers to target registers
  - spill registers if needed
- if target registers < source registers
  - map some to memory
  - map on per-block basis
- Reduces load/store significantly
  - improves performance



Register Mapping (2)

```
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)
```

Predecoding Vs. Binary Translation

- Requirement of interpretation routines during predecoding.
- After binary translation, code can be directly executed.

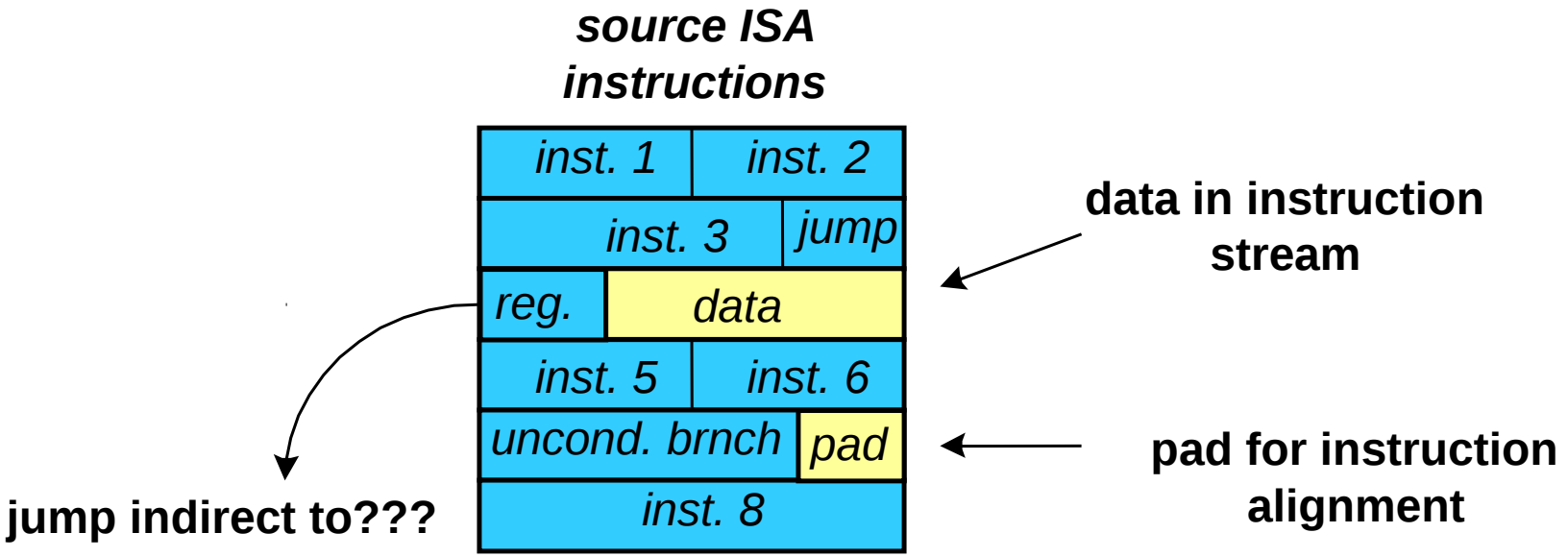
Code Discovery Problem

- May be difficult to *statically* translate or predecode the entire source program
- Consider x86 code

```
                 | mov  %ch,0 ??
31 c0 | 8b | b5 00 00 03 08 8b bd 00 00 03 00
         | movl  %esi, 0x08030000(%ebp)  ??
```

Code Discovery Problem (2)

- Contributors to code discovery problem
  - variable-length (*CISC)* instructions
  - indirect jumps
  - data interspersed with code
  - padding instructions to align branch targets



Code Location Problem

- Mapping of the source program counter to the destination PC for indirect jumps
  - indirect jump addresses in the translated code still refer to source addresses for indirect jumps

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
```

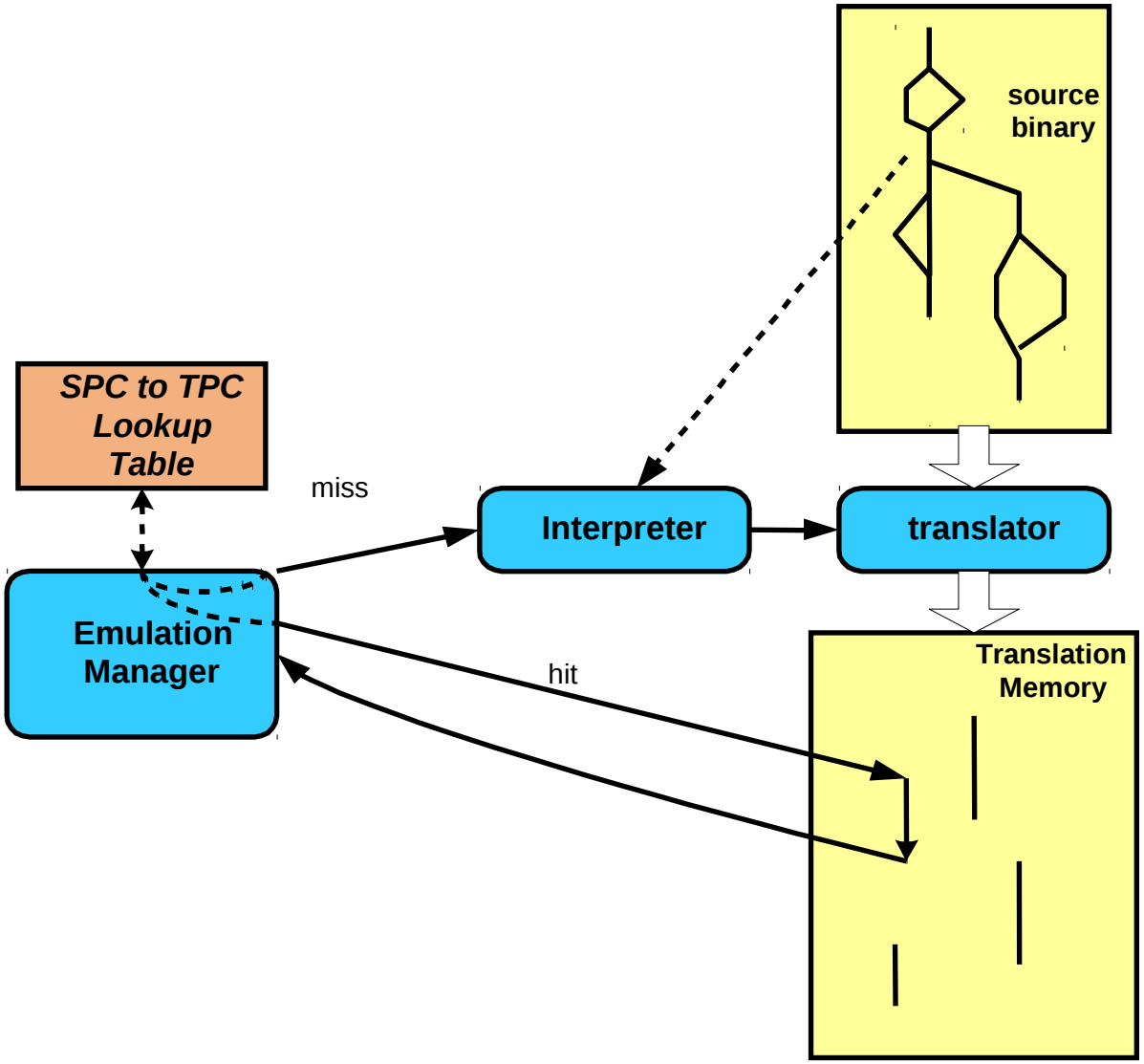
Simplified Solutions

- Fixed-width RISC ISA are always aligned on fixed boundaries
- Use special instruction sets (Java)
  - no jumps/branches to arbitrary locations
  - no data or pads mixed with instructions
  - all code can then be discovered
- Use incremental dynamic translation

Incremental Code Translation

- First interpret
  - perform code discovery as a by-product
- Translate code
  - incrementally, as it is discovered
  - place translated code in *code cache*
  - use lookup table to save source to target PC mappings
- Emulation process
  - execute translated block
  - lookup next source PC in lookup table
    - if translated, jump to target PC
    - else, interpret and translate

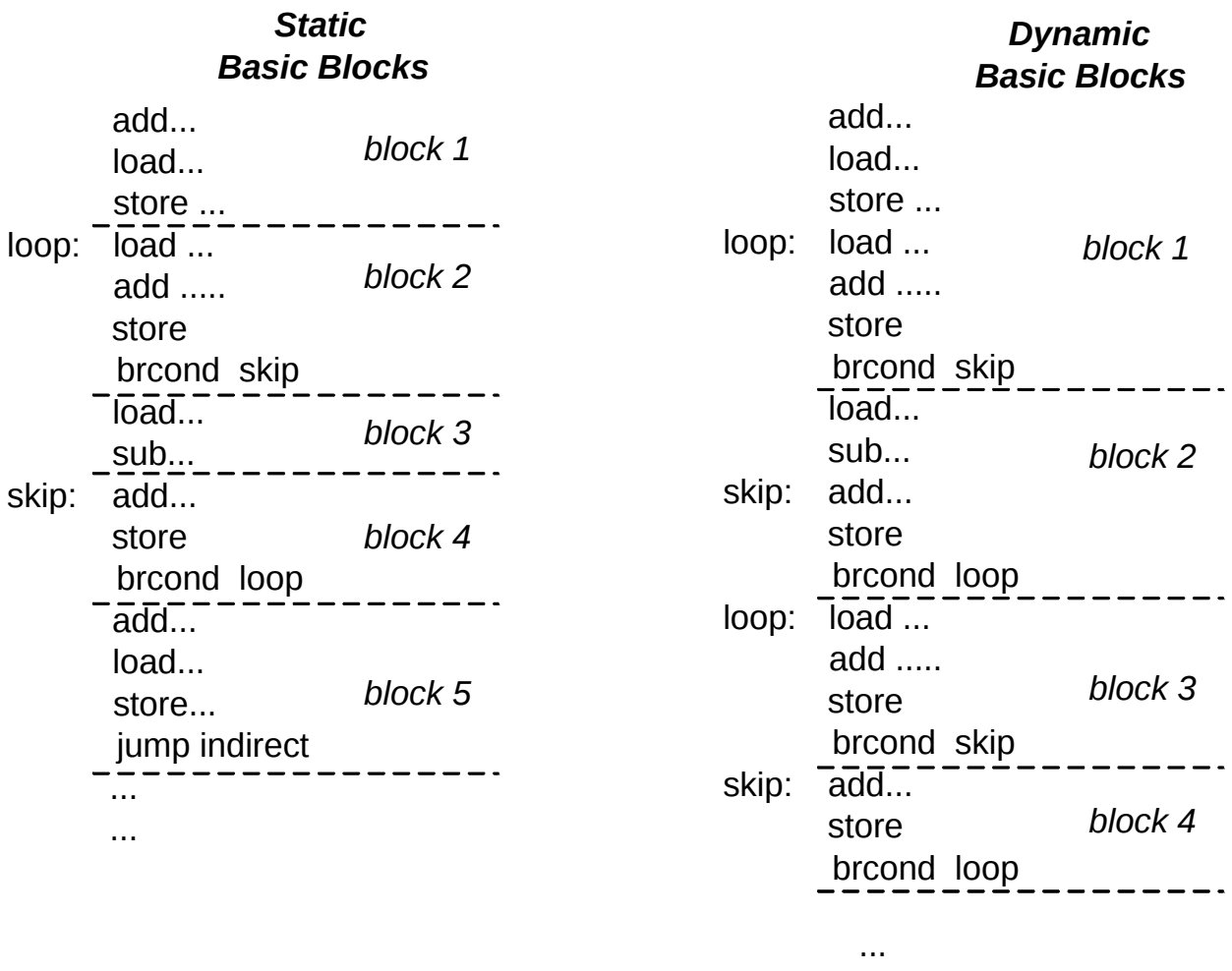
Incremental Code Translation (2)



Dynamic Basic Block

- Unit of translation during dynamic translation.
- *Leaders* identify starts of static basic blocks
  - first program instruction
  - instruction following a branch or jump
  - target of a branch or jump
- Runtime control flow identify dynamic blocks
  - instruction following a taken branch or jump at runtime

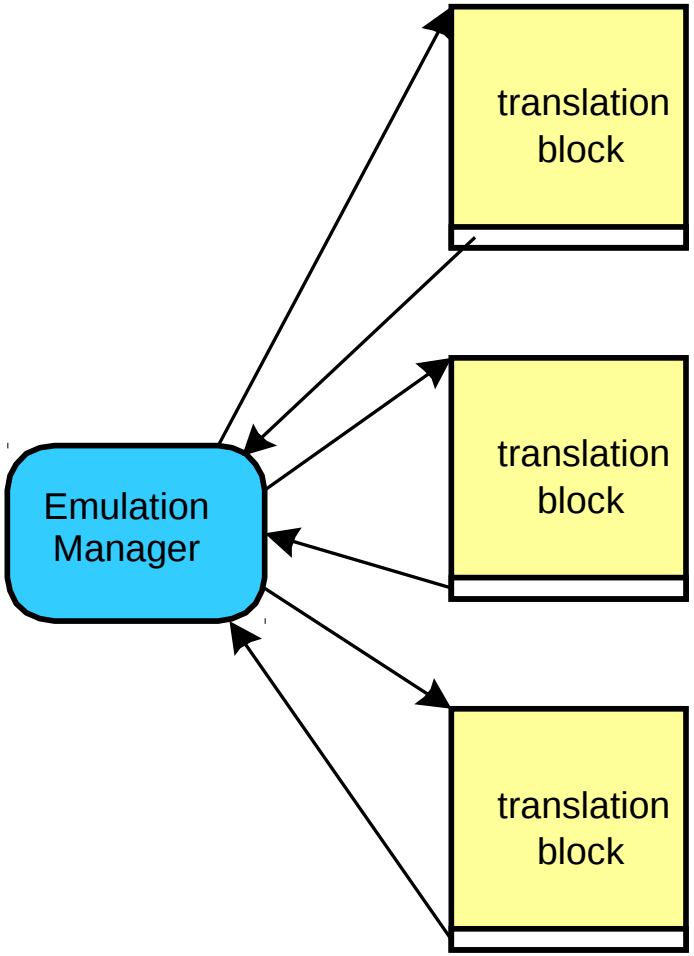
Dynamic Basic Block (2)



Flow of Control

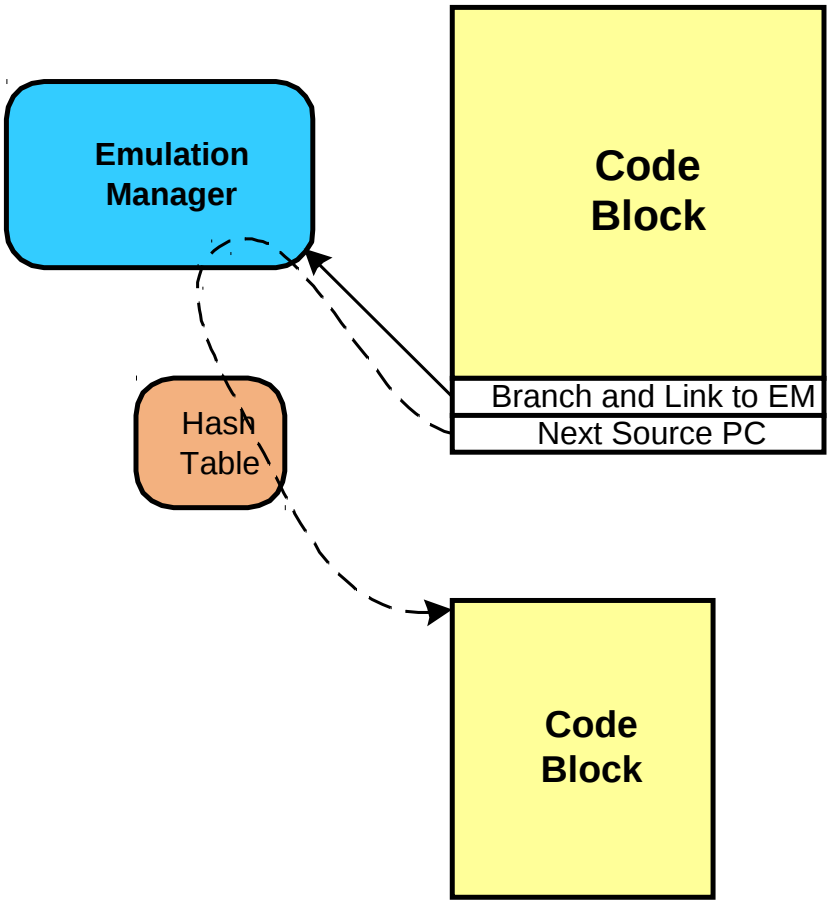
- Even after all blocks are translated, control flows between translated blocks and emulation manager.
- EM connects the translated blocks during execution.
- Optimizations can reduce the overhead of going through the EM between every pair of translation blocks.

Flow of Control (2)

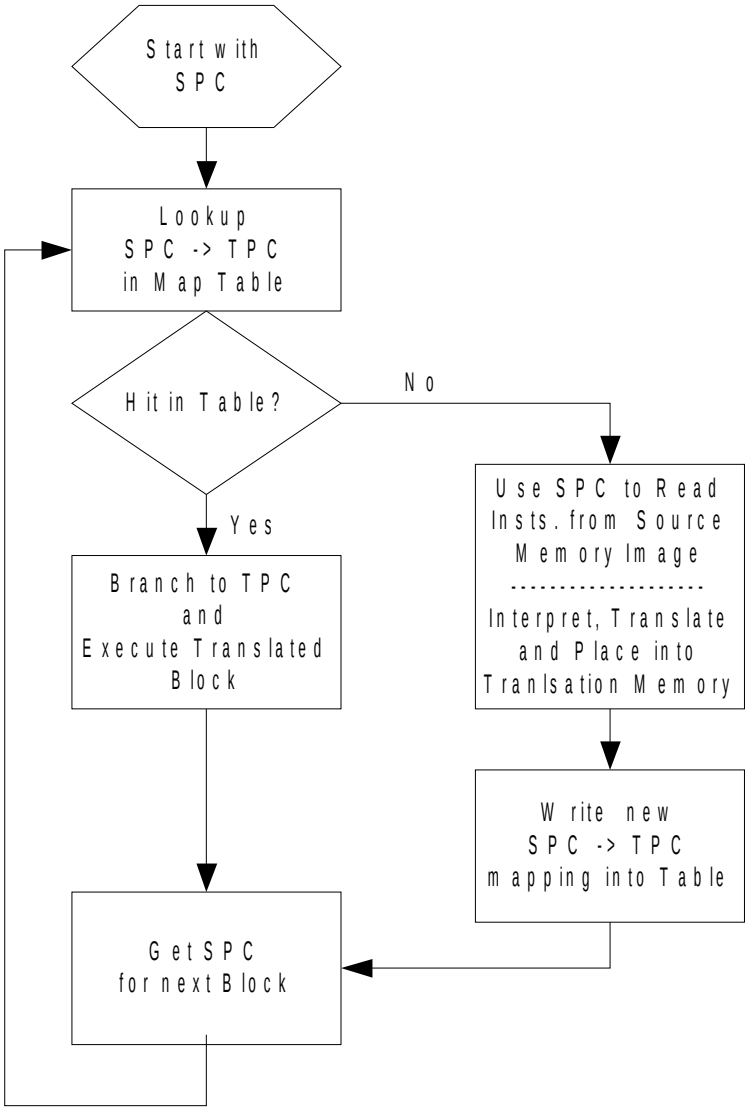


Tracking the Source PC

- Update SPC as part of translated code
  - place SPC in stub
- General approach
  - translator returns to EM via *branch-and-link (BL)*
  - SPC placed in stub immediately after BL
  - EM uses link register to find SPC and hash to next target code block



Emulation Manager Flowchart

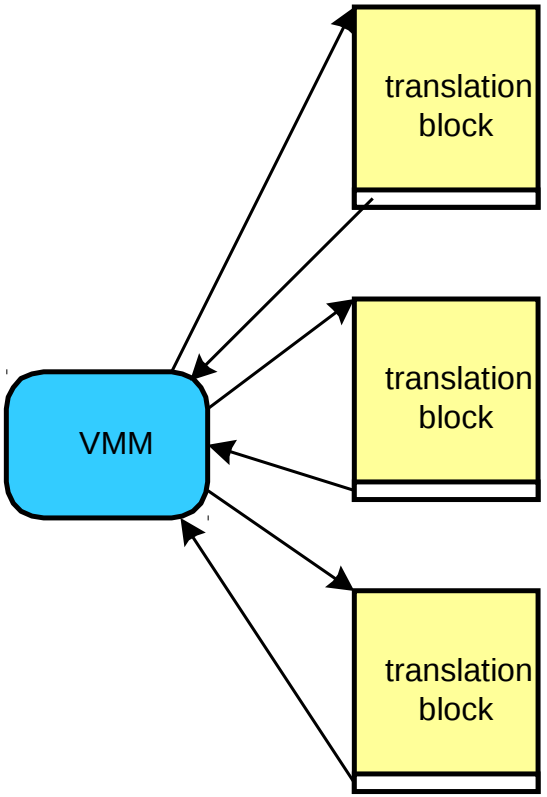


Translation Chaining

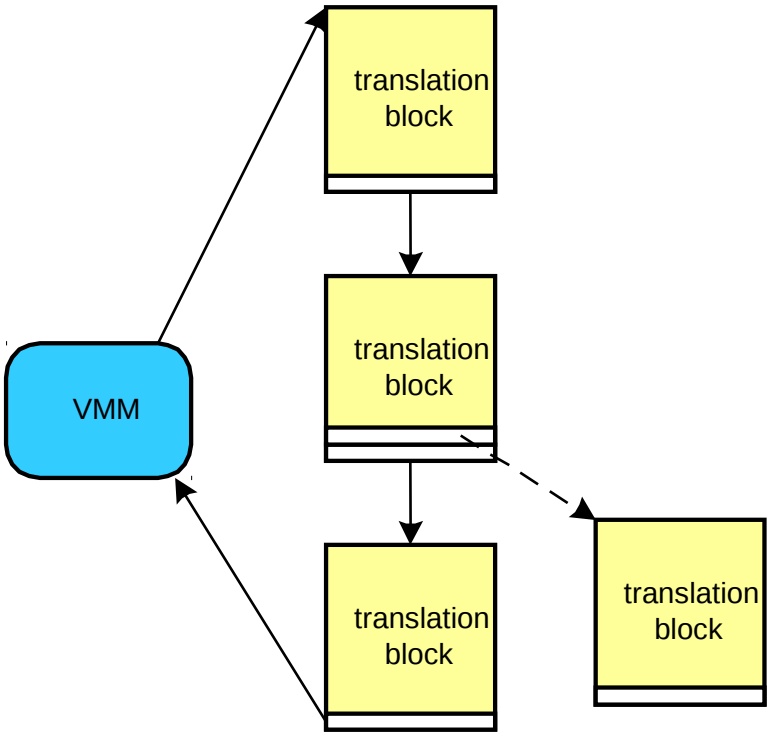
- Translation blocks are linked into chains
- If the successor block has not yet being translated
  - code is inserted to jump to the EM
  - later, after jumping to the EM, if the EM finds that the successor block has being translated, then the jump is modified to instead point directly to the successor

Translation Chaining (2)

Without Chaining

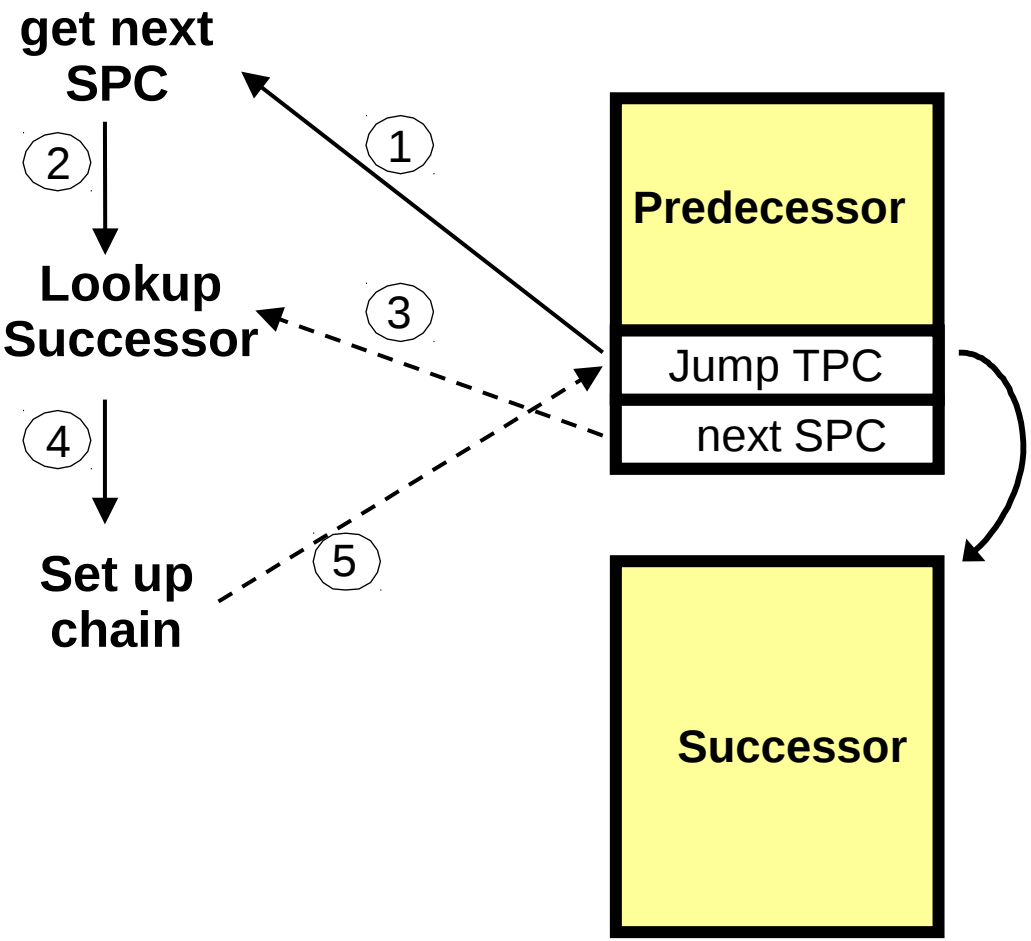


With Chaining



Translation Chaining (3)

- Creating a link:



Translation Chaining (4)

PowerPC Translation

```
9AC0:   lwz     r16,0(r4)       ;load value from memory
        add     r7,r7,r16       ;accumulate sum
        stw     r7,0(r5)        ;store to memory
        addic.  r5,r5,-1        ;decrement loop count, set cr0
        beq     cr0,pc+12       ;branch if loop exit
        bl      F000            ;branch & link to EM
        4FDC                    ;save source PC in link register
9AE4:   b       9c08            ;branch along chain
        51C8                    ;save source PC in link register

9C08:   stw     r7,0(r6)        ;store last value of %edx
        xor     r7,r7,r7        ;clear %edx
        bl      F000            ;branch & link to EM
        6200                    ;save source PC in link register
```

Software Indirect Jump Prediction

- For blocks ending with an indirect jump
  - chaining cannot be used as destination can change
  - SPC–TPC map table lookup is expensive
- indirect jump locations seldom change
  - use *profiling* to find the common jump addresses
  - *inline* frequently used SPC addresses; most frequent SPC destination addresses given first

```
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
```

Dynamic Translation Issues

- Tracking the source PC
  - SPC used by the emulation manager and interpreter
- Handle self-modifying code
  - programs modifying (perform stores) code at runtime
- Handle self-referencing code
  - programs perform loads from the source code
- Provide precise traps
  - provide precise source state at traps and exceptions

Same – ISA Emulation

- Same source and target ISAs
- Applications
  - simulation
  - OS call emulation
  - program shepherding
  - performance optimization

Instruction Set Issues

- Register architectures
  - register mappings, reservation of special registers
- Condition codes
  - lazy evaluation as needed
- Data formats and arithmetic
  - floating point
  - decimal
  - MMX
- Address resolution
  - byte vs word addressing
- Data Alignment
  - natural vs arbitrary
- Byte order
  - big/little endian

Register Architectures

- GPRs of the target ISA are used for
  - holding source ISA GPR
  - holding source ISA special-purpose registers
  - point to register context block and memory image
  - holding intermediate emulator values
- Issues
  - target ISA registers < source ISA registers
  - prioritizing the use of target ISA registers

Condition Codes

- Condition codes are not used uniformly
  - IA-32 ISA sets CC implicitly
  - SPARC and PowerPC set CC explicitly
  - MIPS ISA does not use CC
- Neither ISA uses CC
  - nothing to do
- Source ISA does not use CC, target ISA does
  - easy; additional ins. to generate CC values

Condition Codes (cont…)

- Source ISA has explicit CC, target ISA no CC
  - trivial emulation of CC required
- Source ISA has implicit CC, target ISA no CC
  - very difficult and time consuming to emulate
  - CC emulation may be more expensive than instruction emulation

Condition Codes (cont…)

- Lazy evaluation
  - CC are seldom used
  - only generate CC if required
  - store the operands and the operation that set each condition code
- Optimizations can also be performed to analyze code to detect cases where CC generated will never be used

Lazy Condition Code Evaluation

```
   add  %ecx,%ebx
   jmp  label1
      . . .
label1:  jz     target

R4 ↔ eax        PPC to
R5 ↔ ebx        x86 register
R6 ↔ ecx        mappings
.
.
R24 ↔ scratch register used by emulation code
R25 ↔ condition code operand 1   ;registers
R26 ↔ condition code operand 2   ;used for
R27 ↔ condition code operation   ;lazy condition
                                 ;emulation code
R28 ↔ jump table base address
```

Lazy Condition Code Evaluation (2)

```
    mr      r25,r6          ;save operands
    mr      r26,r5          ;and opcode for
    li      r27,“add”       ;lazy condition code emulation
    add     r6,r6,r5        ;translation of add
    b       label1
            ...
label1:
    bl      genZF           ;branch and link genZF code
    beq     cr0,target      ;branch on condition flag
            ...
genZF:
    add     r29,r28,r27     ;add “opcode” to jump table base
    mtctr   r29             ;copy to counter register
    bctr                    ;branch via jump table
            ...             ...
“add”: add. r24,r25,r26       ;perform PowerPC add, set cr0
       blr                    ;return
```

Data Formats and Arithmetic

- Maintain compatibility of data transformations.
- Data formats are arithmetic operations are standardized
  - two's complement representation
  - IEEE floating point standard
  - basic logical/arithmetic operations are mostly present
- Exceptions:
  - IA32 FP uses 80-bit intermediate results
  - PowerPC and HP PA have multiply-and-add (FMAC) which has a higher precision on intermediate values
  - integer divide vs. using FP divide to approximate
- ISAs may have different immediate lengths

Memory Address Resolution

- ISAs can access data items of different sizes
  - load / stores of bytes, halfwords, full words, as opposes to only bytes and words
- Emulating a less powerful ISA
  - no issue
- Emulating a more powerful ISA
  - loads: load entire word, mask un-needed bits
  - stores: load entire word, insert data, store word

Memory Data Alignment

- Aligned memory access
  - word accesses performed with two low order bits 00, halfword access must have lowest bit 0, etc.
- Target ISA does not allow unaligned access
  - break up all accesses into byte accesses
  - ISAs provide supplementary instructions to simplify unaligned accesses
  - unaligned access traps, and then can be handled

Byte Order

- Ordering of bytes within a word may differ
  - *little endian* and *big endian*
- Target code must perform byte ordering
- Guest data image is generally maintained in the same byte order as assumed by the source ISA
- Emulation software modifies addresses when bytes within words are addressed
  - can be very inefficient
- Some target ISAs may support both byte orders
  - e.g., MIPS, IA-64