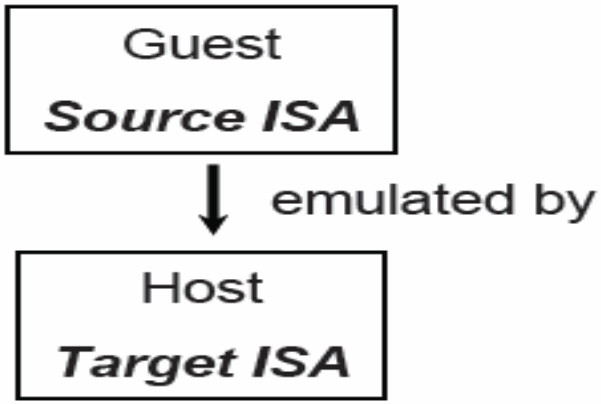
Emulation: Interpretation

Contents

- Emulation, Basic Interpretation
- Threaded Interpretation
- Emulation of a Complex Instruction Set

Emulation

- "Implementing the interface/functionality of one system on a system with different interface/functionality"
- In VM, it means instruction set emulation
  - Implementing one ISA (the *target*) reproduces the behavior of software compiled to another ISA (the *source*)

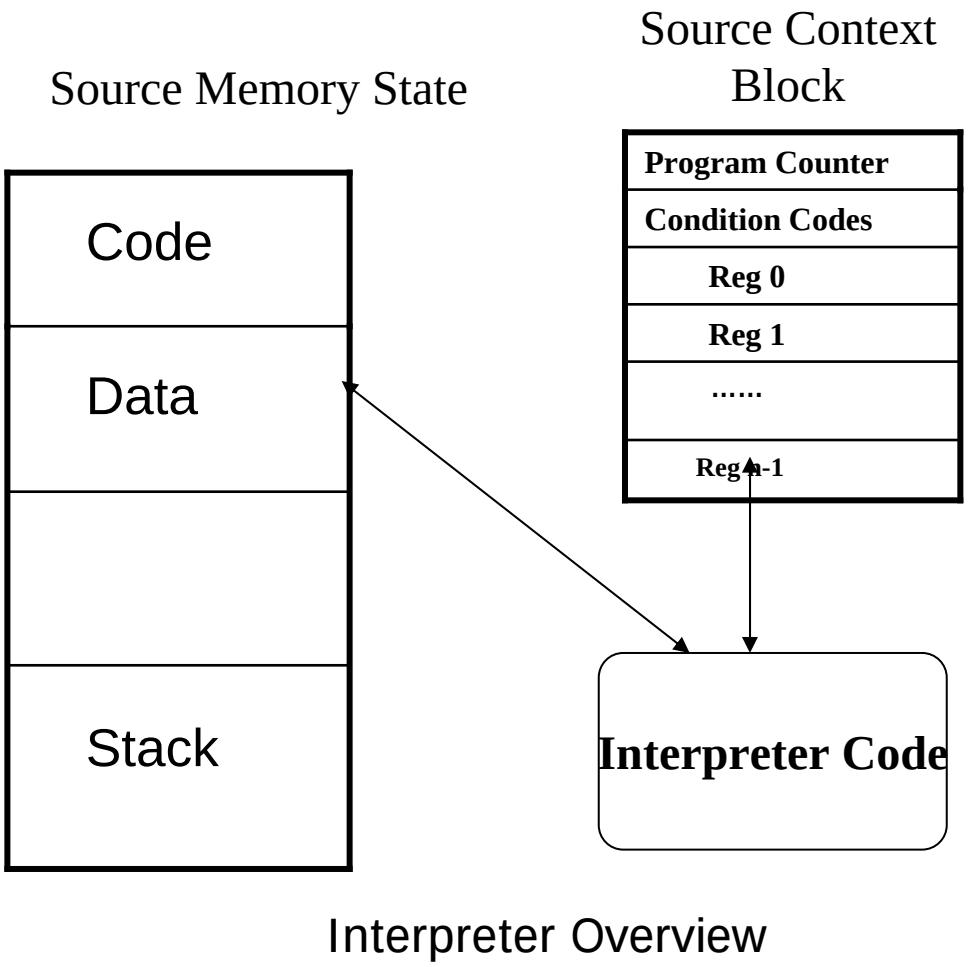


Emulation Methods

- Two methods of emulation: interpretation & binary translation
- Interpretation
  - Repeats a cycle of fetch a source instruction, analyze, perform
- Binary translation
  - Translates a block of source instr. to a block of target instr.
  - Save the translated code for repeated use
  - Bigger initial translation cost with smaller execution cost
  - More advantageous if translated code is executed frequently
- Some in-between techniques
  - Threaded interpretation
  - Predecoding

Basic Interpreter

- Emulates the whole source machine state
- Guest memory and context block is kept in interpreter's memory (heap)
- code and data
- general-purpose registers, PC, CC, control registers



Interpreter Overview

Decode-and-dispatch interpreter

Interpretation repeats

- Decodes an instruction
- Dispatches it to an interpretation routine based on the type of instruction
- Code for interpreting PPC ISA

```
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 Functions

- Emulation of the LWZ instruction

```
void 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;
}
```

Instruction Functions

- Emulation of the ALU instruction(s)

```
void 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;
}
```

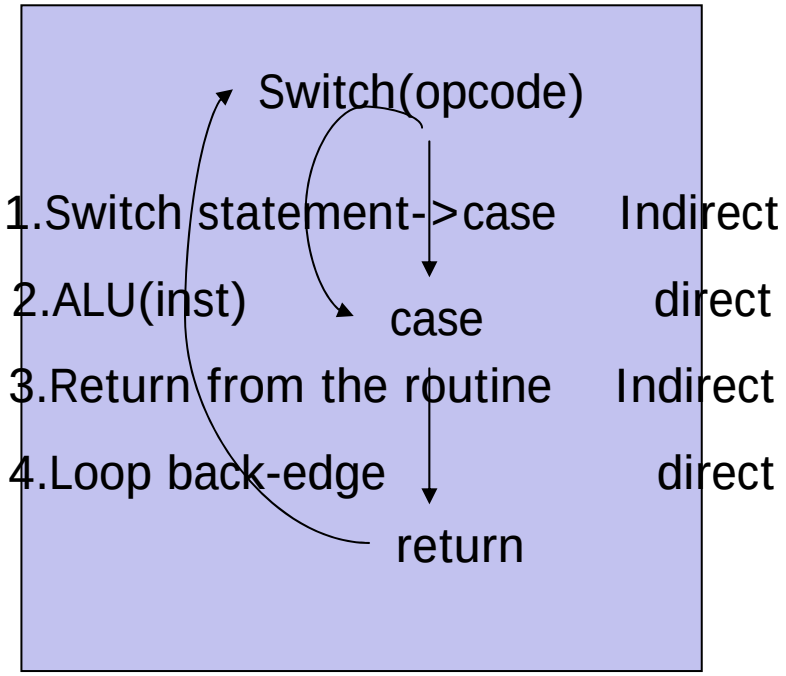
Decode-and-dispatch interpreter

- Advantage
  - Low memory requirements
  - Zero star-up time
- Disadvantage:
  - Steady-state performance is slow
  - A source instruction must be parsed each time it is emulated
  - Lots of branches would degrade performance

- How many branches are there in our interpreter code?

Branches in Decode-&-Dispatch

```
While (!halt&&interrupt){
    switch(opcode){
     case ALU:ALU(inst);
              . . . . .
}
```



We can remove all of these branches with threading

Threaded Interpretation: Idea

- **Put the dispatch code to the end of each interpretation routine.**

```
Instruction function list
    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;
}
```

Threaded Interpretation (2)

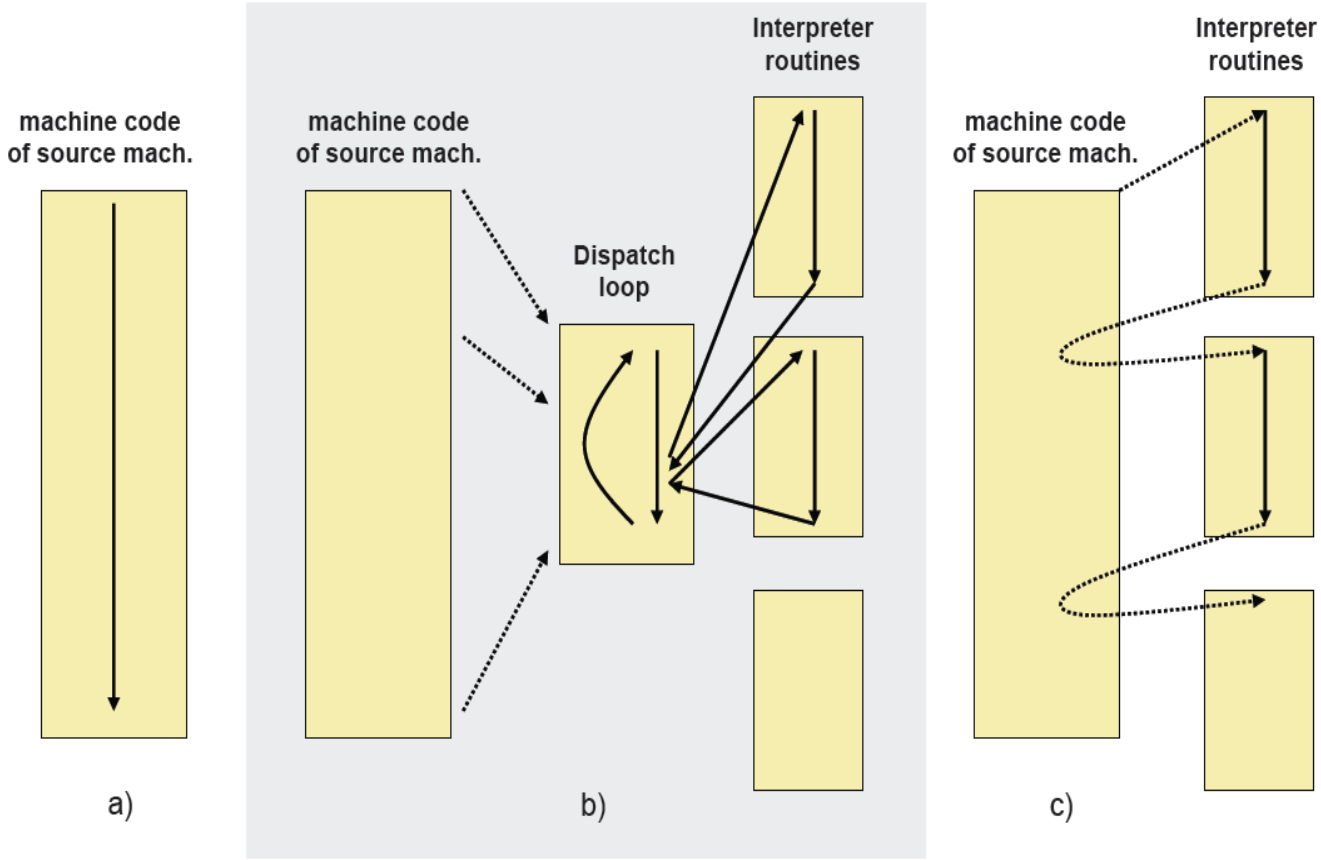
- Solution:
  - Append part of the dispatch code to the end of each instruction interpretation routine

```
void 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;
        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;
```

```
void 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(ins
                case AddExtended: AddExtended(
                ...
        }
        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;
```

Threaded Interpretation (3)

- Control flow in various modes of execution:
  - a) native, b) DnD interpreter, c) threaded interpretation



Threaded Interpretation

- One key point is that dispatch occurs indirectly thru a dispatch table

```
routine = dispatch[opcode,extended_opcode];
  goto *routine;
```

- Also called indirect threaded interpretation
- Then, what would be directed threaded interpretation?
- Can we remove the overhead of accessing the table?
- Solution: predecoding and direct threading

Predecoding

- Extracting various fields of an instruction is complicated
  - Fields are not aligned, requiring complex bit extraction
  - Some related fields needed for decoding is not adjacent
- If it is in a loop, this extraction job should be repeated

- How can we reduce this overhead? Predecoding
  - Pre-parsing instructions in a form that is easier to interpreter
  - Done before interpretation starts
  - Predecoding allows direct threaded interpretation

Predecoding for PPC

- In PPC, opcode & extended opcode field are separated and register specifiers are not byte-aligned
- Define instruction format and define an predecode instruction array based on the format

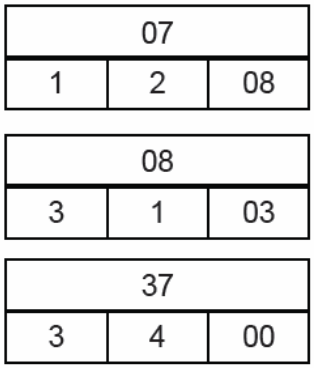
```
Struct instruction {
              unsigned  long op;   // 32 bit
              unsigned  char dest; // 8  bit
              unsigned  char src1; // 8  bit
              unsigned  int  src2; // 16 bit
} code [CODE_SIZE];
```

- Pre-decode each instruction based on this format

Predecoding Example

- Example: Code to accumulate a value, e.g. as a loop body

```
lwz     r1, 8(r2)   ; load word and zero
add     r3, r3, r1  ; r3 = r3 + r1
stw     r3, 0(r4)   ; store word
```



(load word and zero)

(add)

(store word)

Previous Interpreter Code

- It is more efficient to perform the repeated decoding operations only once per source machine address

```
void 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;
}
```

New Interpreter Code

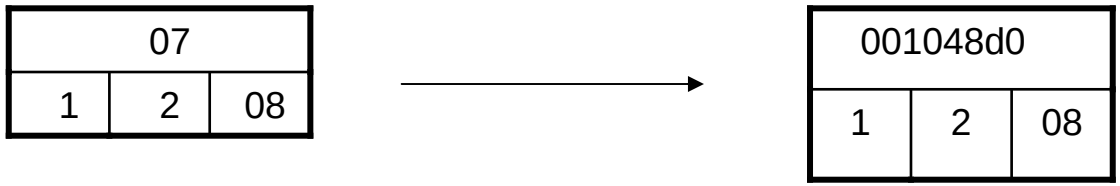
- These predecoded instructions (contained in an array) can now be executed by the following code:

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

```
LoadWordAndZero:
        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;
```

Directed Threaded Interpretation

- Even with predecoding, indirect threading includes a centralized dispatch table, which requires
  - Memory access and indirect jump
- To remove this overhead, replace the instruction opcode in predecoded format by address of interpreter routine



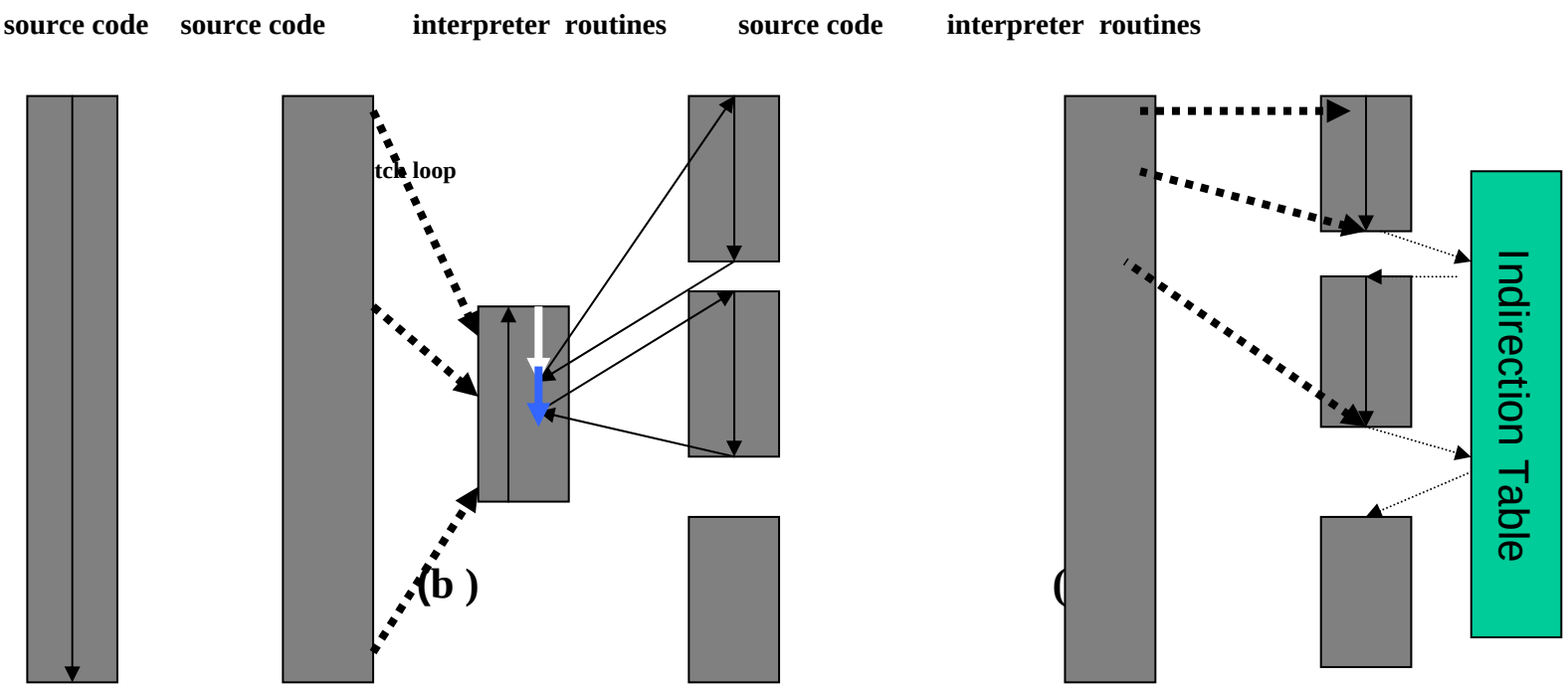

```
If (halt || interrupt) goto exit;
opcode= code[TPC].op;
routine=dispatch [opcode];
goto  *routine;
```

→

```
If (halt || interrupt) goto exit;
routine= code[TPC].op;
goto  *routine;
```

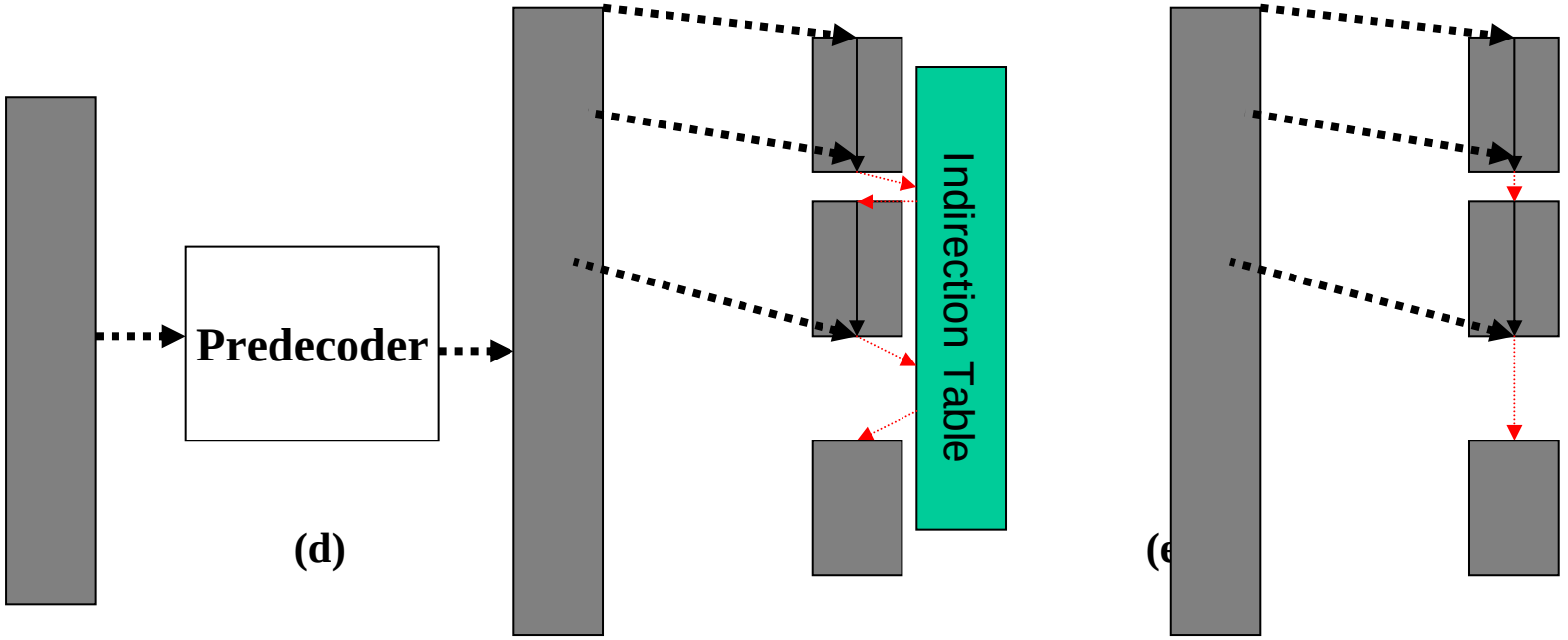
Comparison

Dispatch-&-Decode Indirect Threaded



Comparison

Predecoded Indirect Threaded **Direct Threaded**



Comparison

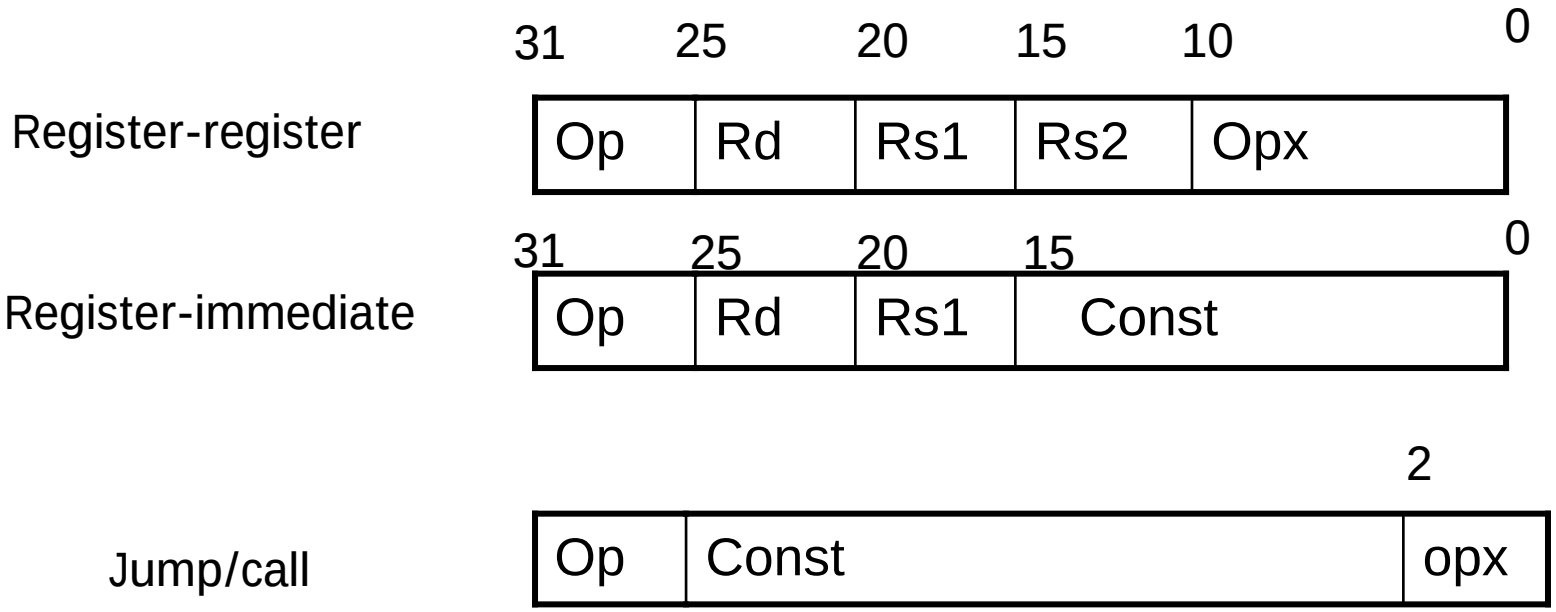
	Decode-and-Dispatch	Indirect Threaded Interpreter	Direct Threaded Interpreter
Memory requirements	Low	Low	High
Start-up performance	Fast	Fast	Slow
Steady-state performance	Slow	Slow (better than the first one)	Medium
Code portability	Good	Good	Medium

DSVM

- Dynamic Samsung Virtual Machine
- Splitted interpreter
  - Inner, Outer loop
  - Instruction cache
- Indirect threaded interpretation

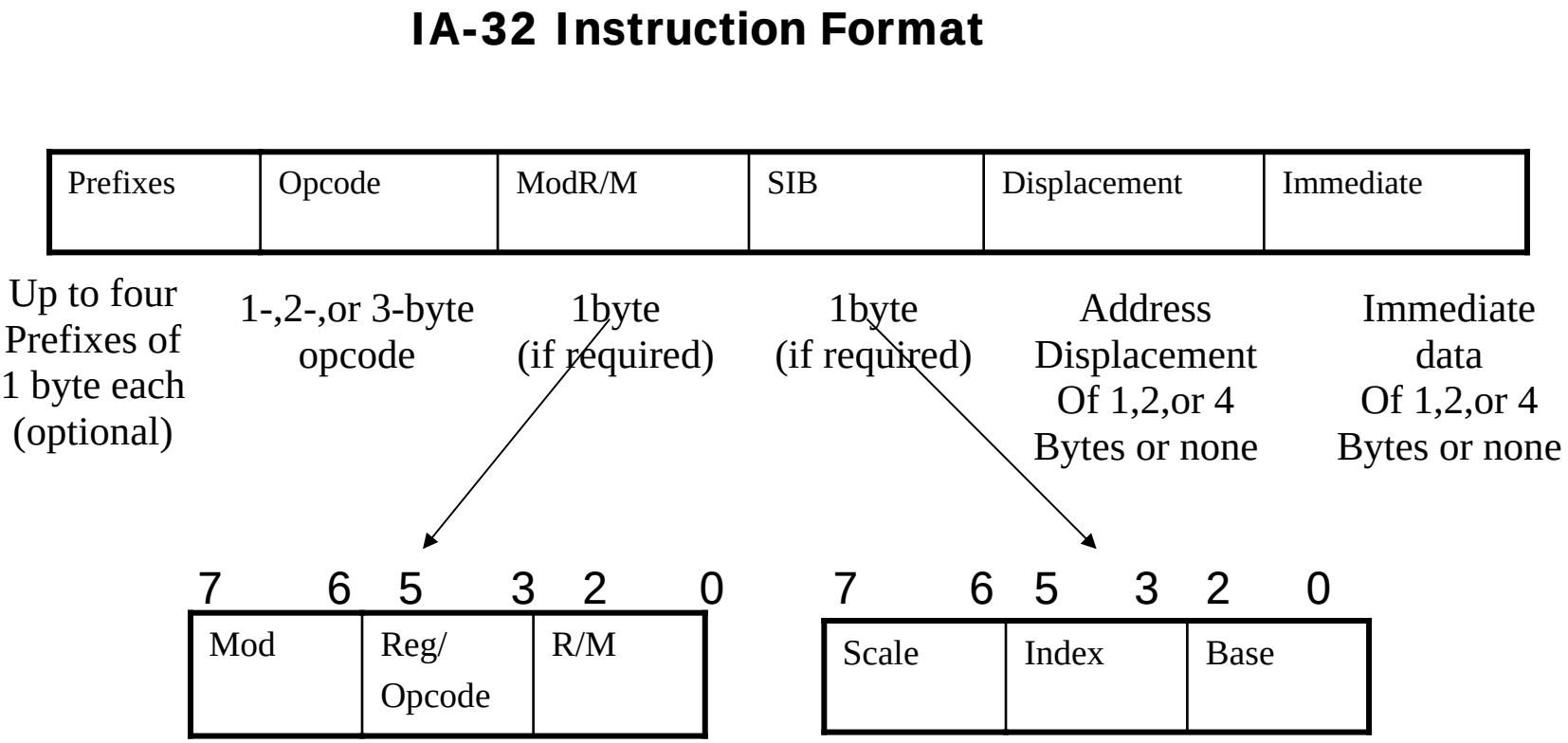
Interpreting CISC ISA

- RISC ISA (Power PC) 32 bit register. 32bit length.



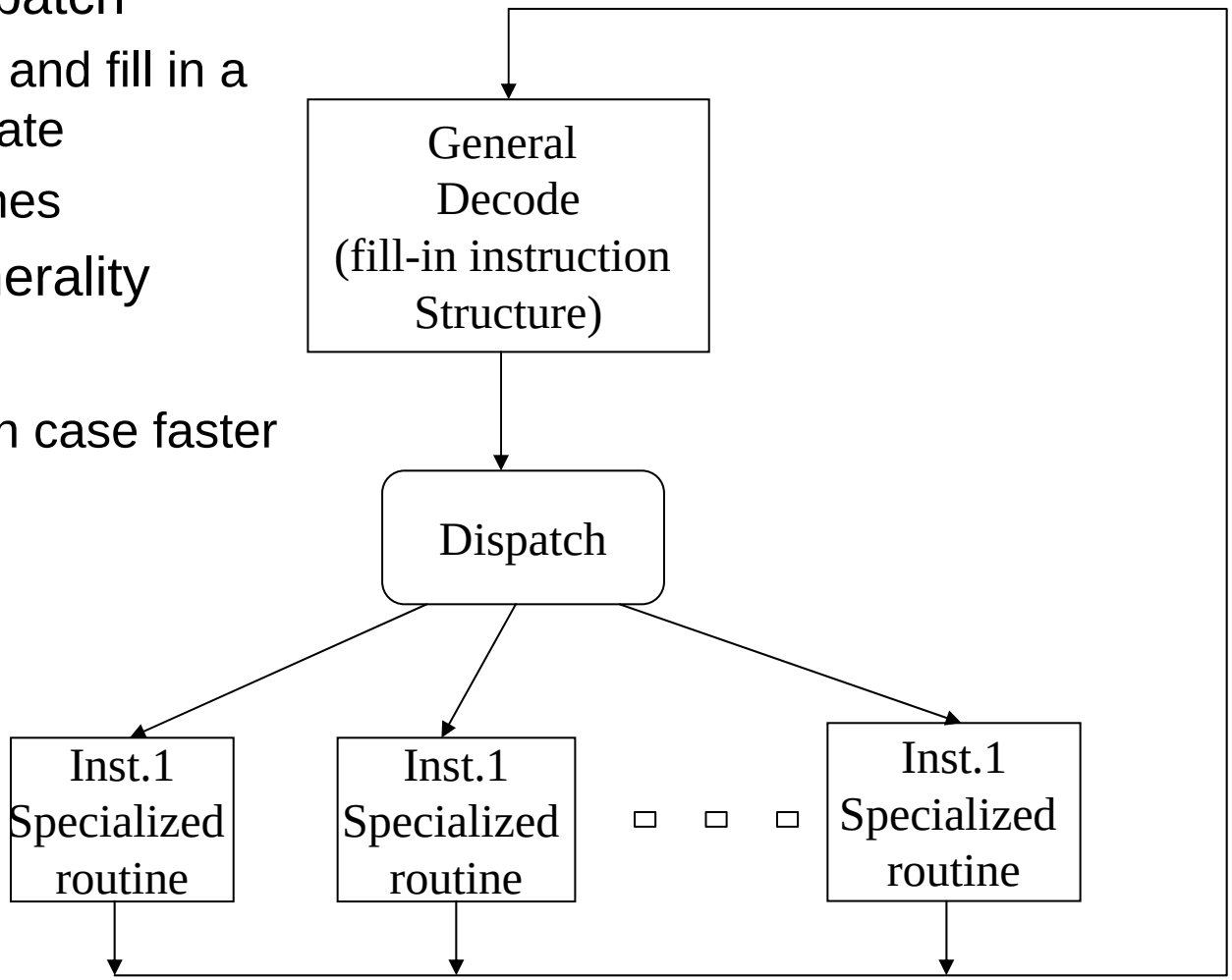
Interpreting a Complex Instruction Set

CISC instruction set has a wide variety of formats, variable instruction lengths, and variable field lengths (x86 instruction lengths: 1 ~ 16 bytes)

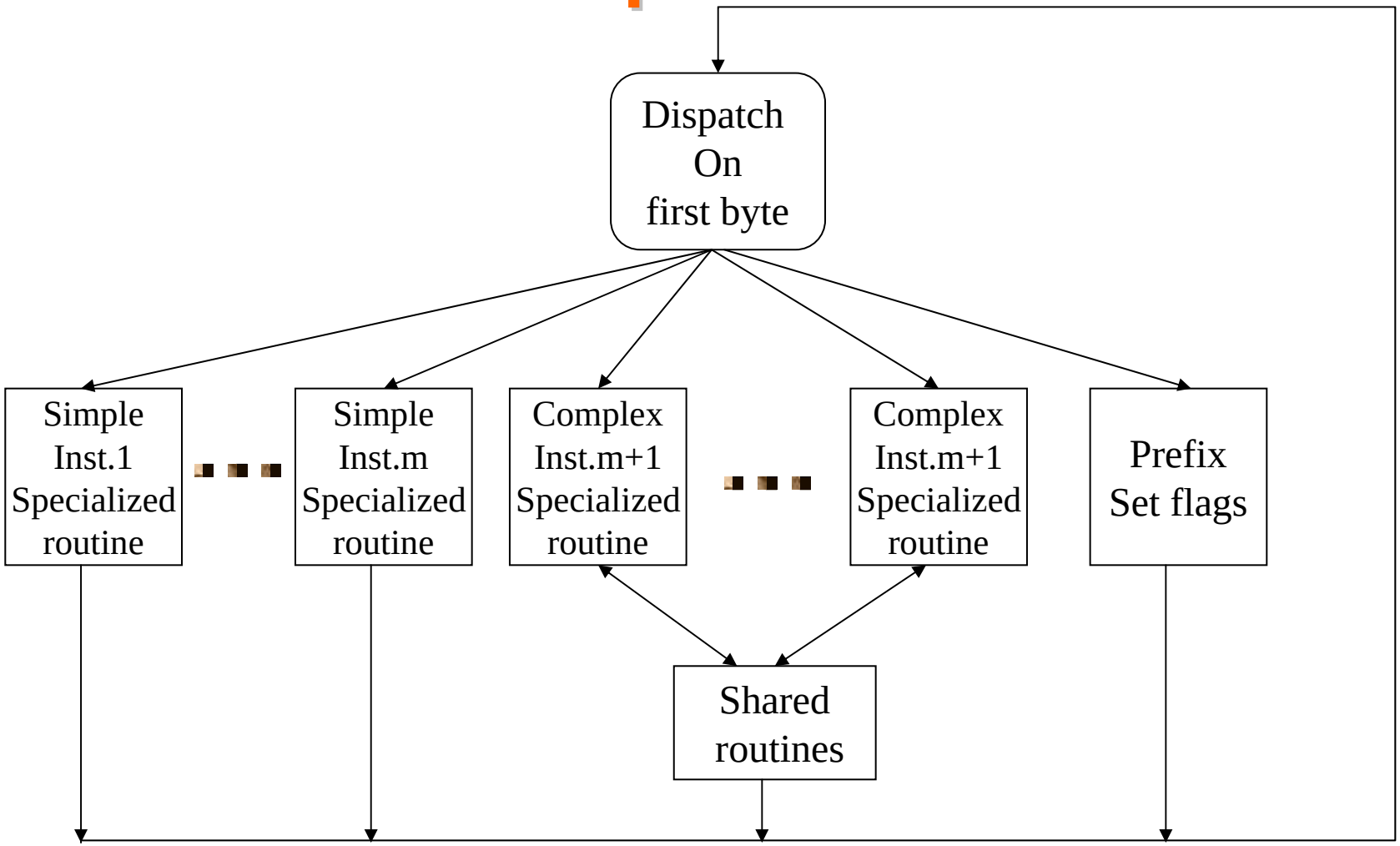


Interpreting a Complex Instruction Set

- Decode and dispatch
  - Decode fields and fill in a general template
  - Jump to routines
- Slow due to generality
- Solution
  - Make common case faster



Some optimizations



Threaded Interpretation