Operating Systems Design (CS 423)

Elsa L Gunter

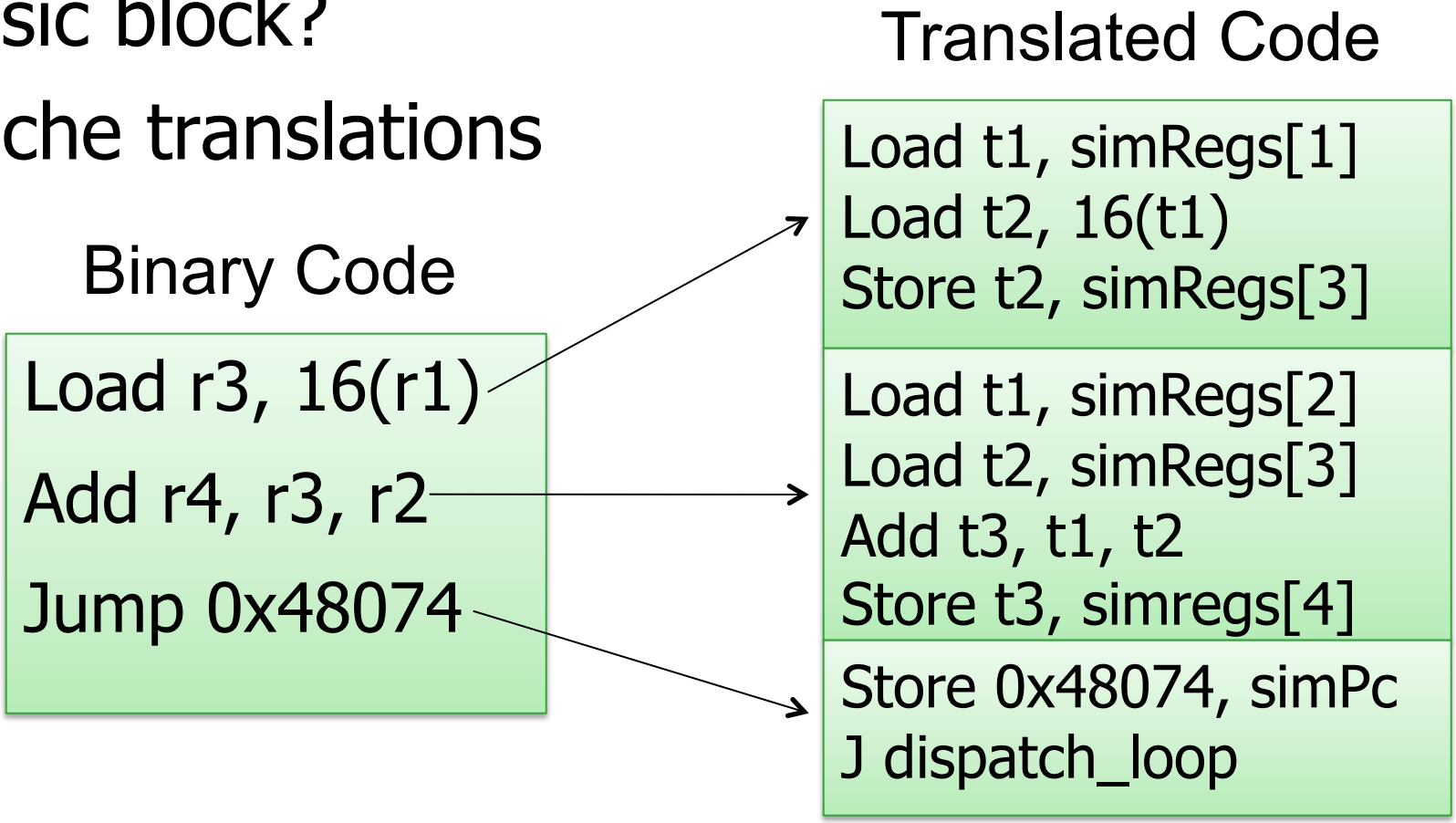
2112 SC, UIUC

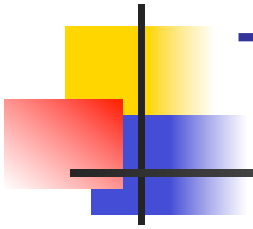
http://www.cs.illinois.edu/class/cs423/

Based on slides by Roy Campbell, Sam King, and Andrew S Tanenbaum

Dynamic binary translation

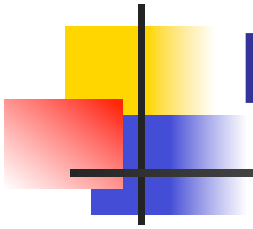
- Simulate basic block using host instructions. Basic block?
- Cache translations





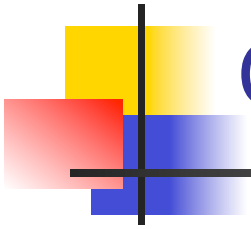
This is Binary?

- We are calling the translation on previous slide dynamic “binary” translation.
- Where’s the binary?
- Answer, part 1: Decode translates binary effectively to assembly



Decode

```
void decode_control_signals(u32 opcode, struct
   control_signals *control) {
     control->raw = opcode;
     control->op = (opcode >> 30) & 0x3;
     control->rd = (opcode >> 25) & 0x1f;
     control->op2 = (opcode >> 22) & 0x7;
     control->op3 = (opcode >> 19) & 0x3f;
     control->rs1 = (opcode >> 14) & 0x1f;
     control->i = (opcode >> 13) & 0x1;
     control->asi = (opcode >> 5) & 0xff;
     control->simm = sign_extend_13(opcode & 0x1fff);
     control->imm = opcode & 0x3fffff;
     control->rs2 = opcode & 0x1f;
     control->disp22 = opcode & 0x3fffff;
     control->disp30 = opcode & 0x3fffffff;
     control->cond = (opcode >> 25) & 0xf;
}
```



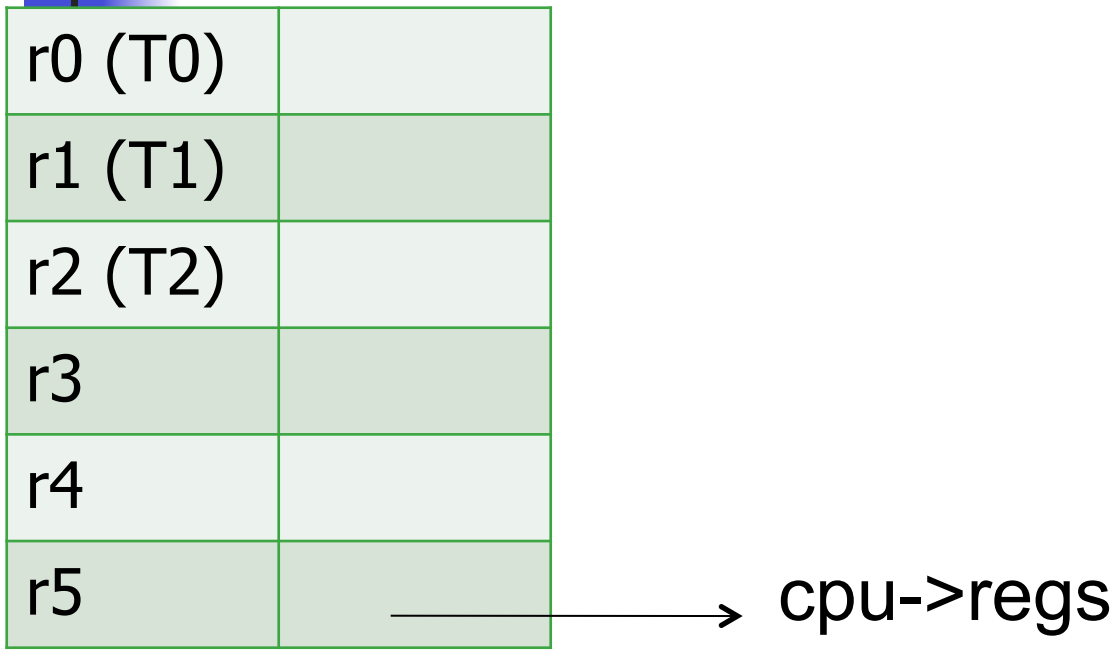
OK, but aren't we generating assembly?

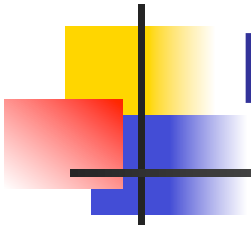
- In slides we showed

  gen_load_T0(guest_reg_no)

  producing "assembly"
- This is an abstraction
- It needs to produce binary, ie bits

Dynamic Binary Translation



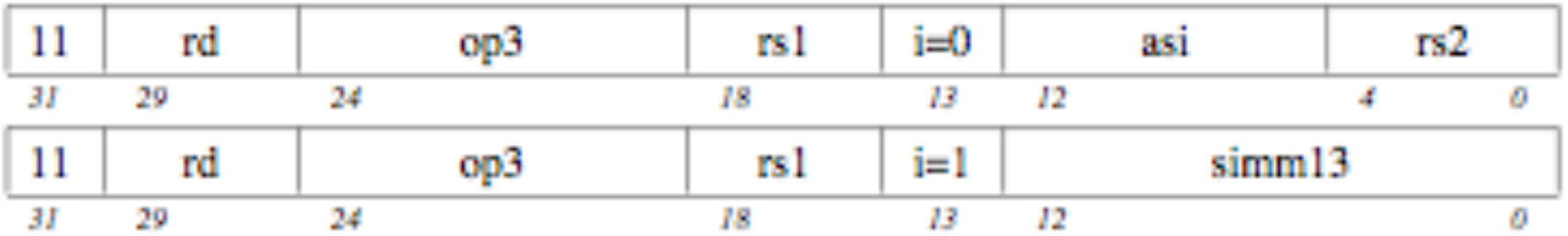


Binary code generation

- What does it mean that
- gen_load_T0(guest_reg_no)
- Produces
- Load r0, gr1_off(r5)?

Finding the pieces: Load r0, gr1_off(r5)

Format (3):

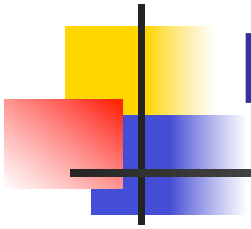


Format	*op*	Instructions
1	1	CALL
2	0	Bicc, FBfcc, CBccc, SETHI
3	3	memory instructions
3	2	arithmetic, logical, shift, and remaining

ASI	Address Space
0x00 – 0x07	implementation-dependent
0x08	User Instruction
0x09	Supervisor Instruction
0x0A	User Data
0x0B	Supervisor Data
0x0C – 0xFF	implementation-dependent

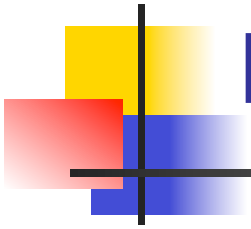
opcode	*op3*	*operation*
LDSB	001001	Load Signed Byte
LDSH	001010	Load Signed Halfword
LDUB	000001	Load Unsigned Byte
LDUH	000010	Load Unsigned Halfword
LD	000000	Load Word
LDD	000011	Load Doubleword

We want op = 3,
op3 = 0 (LD) , i = 1
rd = 0, rs1=5,
simm13= 4 * grn



Binary code generation

```
gen_load_T0(guest_reg_no) {
    u32 op = 0x3 << 30;
    u32 op3 = 0x0 << 19;
    u32 i = 0x1 << 13;
    u32 rd = 0x0 << 25;
    u32 rs1 = 0x5 << 14;
    u32 simm13 = guest_reg_no * 4;
    return op & rd & op3 & rs1 & i & simm13}
```

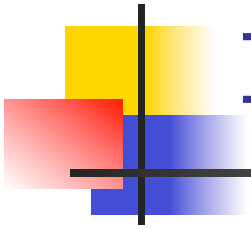


Dynamic binary translation for x86

- Goal: translate this basic block
- Note: x86 assembly dst value is last

```
Mov %rsp, %rbp    //move stack ptr -> frame ptr
Sub $0x10, %rsp   //sub 10 from stack pointer
Movq $0x100000, 0xfffffffffffffff8(%rbp)
//set value into loc rel to frame ptr
```

- Problem: x86 instructions have varying length

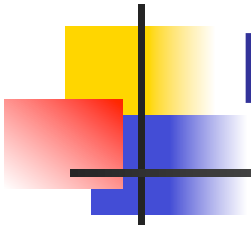


Inst-by-inst simulation: fetch

```
void fetch(struct CPU *cpu, struct control_signals *control){
 memset(control, 0, sizeof(struct control_signals));
 control->orig_rip = cpu->rip;
 control->opcode[control->numOpcode++] = fetch_byte(cpu);
 if(IS_REX_PREFIX(control->opcode[0])) {
  control->rex_prefix =
       control->opcode[control->numOpcode-1];
  control->rex_W = REX_PREFIX_W(control->rex_prefix);
control->opcode[control->numOpcode-1] = fetch_byte(cpu);
}
//and more of this for multi-byte opcodes
```

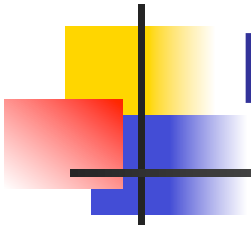
Inst-by-inst simulation: decode

```
if(HAS_MODRM(control->opcode[0]) {
 control->modRM = fetch_byte(cpu);
 control->mod = MODRM_TO_MOD(control->modRM);
 control->rm = MODRM_TO_RM(control->modRM);
 control->reg = MODRM_TO_R(control->modRM);
 // now calculate m addresses
 if(IS_SIB(control)){
    // more address lookup and fetching
 } else if(IS_DISP32) {
   // fetch immediate
 } else { addr = getRegValue(cpu, control) }
// now fetch any disp8 or disp32 for addr
```



Dynamic translation

- Amortize the cost of fetch and decode
- Two different contexts
  - Translator context – fetch and decode
  - Guest context – execute
- Makes debugging hard

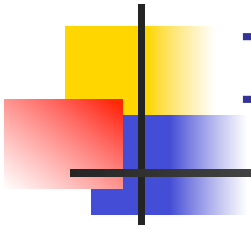


Dynamic translation for x86

- Map guest registers into these host registers

- T0 -> rax
- T1 -> rbx
- T2 -> rcx – use for host addresses

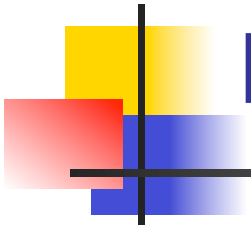
- Rdi -> holds a pointer to cpu



Inst-by-inst: Mov %rsp, %rbp

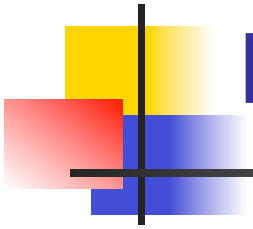
```
cpu->reg[control->rm] =
      cpu->reg[control->reg];
```

Where `rm == rbp index`
and `reg == rsp index`



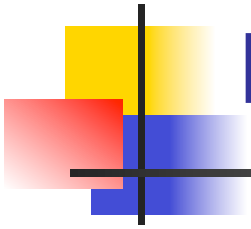
DBT: Mov %rsp, %rbp

```
void mov(struct CPU *cpu,
    struct control_signals *control) {
 gen_load_T0(cpu->tcb,
    getRegOffset(cpu, control->reg));
 gen_store_T0(cpu->tcb,
    getRegOffset(cpu, control->rm));
}
// tcb = translation code block
```



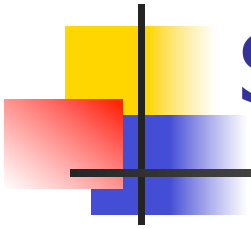
DBT: Mov %rsp, %rbp

```
void gen_load_T0(struct tcb *tcb, uint32_t
  offset) {
uint64_t opcode = offset;
// create movq offset(%rdi), %rax
 opcode <<= 24;
 opcode |= 0x878b48;
// store in cache
 memcpy(tcb->buffer+tcb->bytesInCache,
        translation, size);
 tcb->bytesInCache += size;
```



DBT: compile

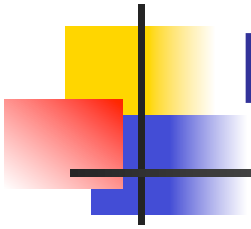
```
// Mov %rsp, %rbp
gen_load_T0(control->reg)
gen_store_T0(control->rm)

//Sub $0x10, %rsp
gen_load_rm32_T0(cpu, control)
gen_alu_imm32_T0(tcb, imm, SUB_ALU_OP)
gen_store_rm32_T0(cpu, control)

// movq $0x100000, -8(%rbp)
gen_mov_imm32_T0(cpu, control->imm)
gen_store_rm64_T0(cpu, control)
```



Switching to guest context

```
Jump_to_tc(cpu) {
//will return a pointer to beginning of TC buf
void (*foo)(struct CPU *) = lookup_tc(cpu->rip)
foo(cpu);
}
```

- What does the preamble for your TCB look like?
- How do we return back to the host (or translator) context?



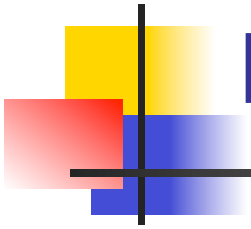
DBT optimizations

Mov %rsp, %rbp

Sub $0x10, %rsp

movq $0x100000, -8(%rbp)

- Are there opportunities for optimizations?



Discussion questions

- Could you run this raw code on the CPU directly?
- What assumptions do you have to make?
- When do you have to invalidate your translation cache
- Break into groups to discuss this
- Read embra for discussion about using guest physical vs guest virtual pc values to index tb