Operating Systems Design (CS 423)

Elsa L Gunter

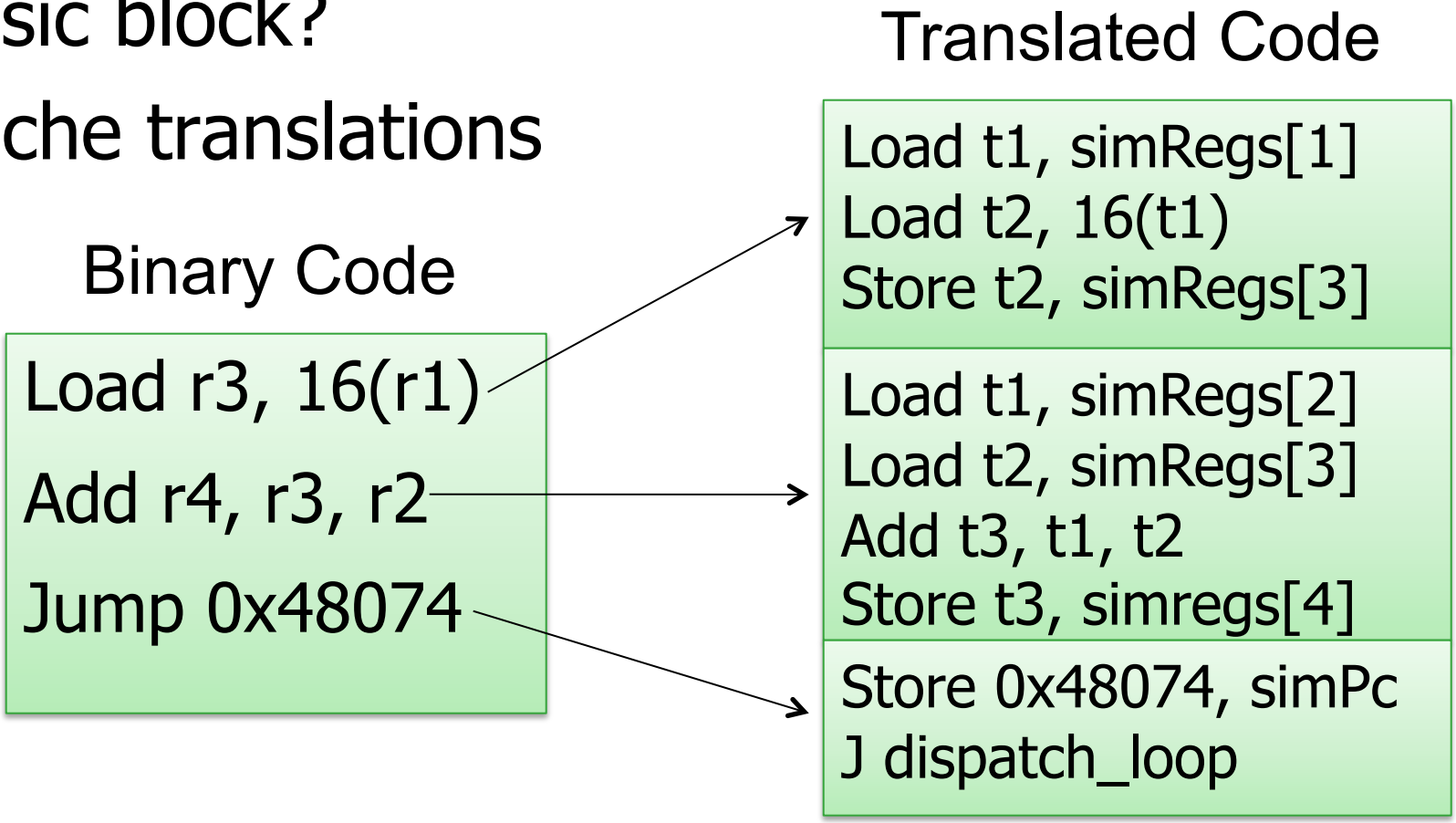
2112 SC, UIUC

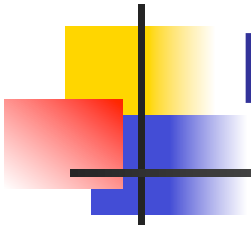
http://www.cs.illinois.edu/class/cs423/

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



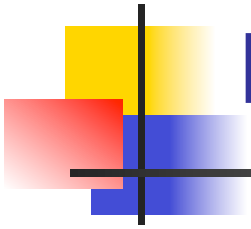


Performance

```
while(1){
 inst = mem[PC]; // fetch
 //decode
 if(inst == add){
 //execute
 ...
 }
} // repeat
```

```
while(1){
 if(!translated(pc)){
  translate(pc);
 }
jump to pc2tc(pc);
} // repeat
```

- Why is this faster?
- Are there disadvantages?



Performance

```
while(1){
 inst = mem[PC]; // fetch
 //decode
 if(inst == add){
 //execute
 ...
 }
} // repeat
```

```
while(1){
 if(!translated(pc)){
  translate(pc);
 }
jump to pc2tc(pc);
} // repeat
```

- Why is this faster?
  - Caching the translation
- Are there disadvantages?
  - Harder to code, slower if only execute once

Dynamic Binary Translation

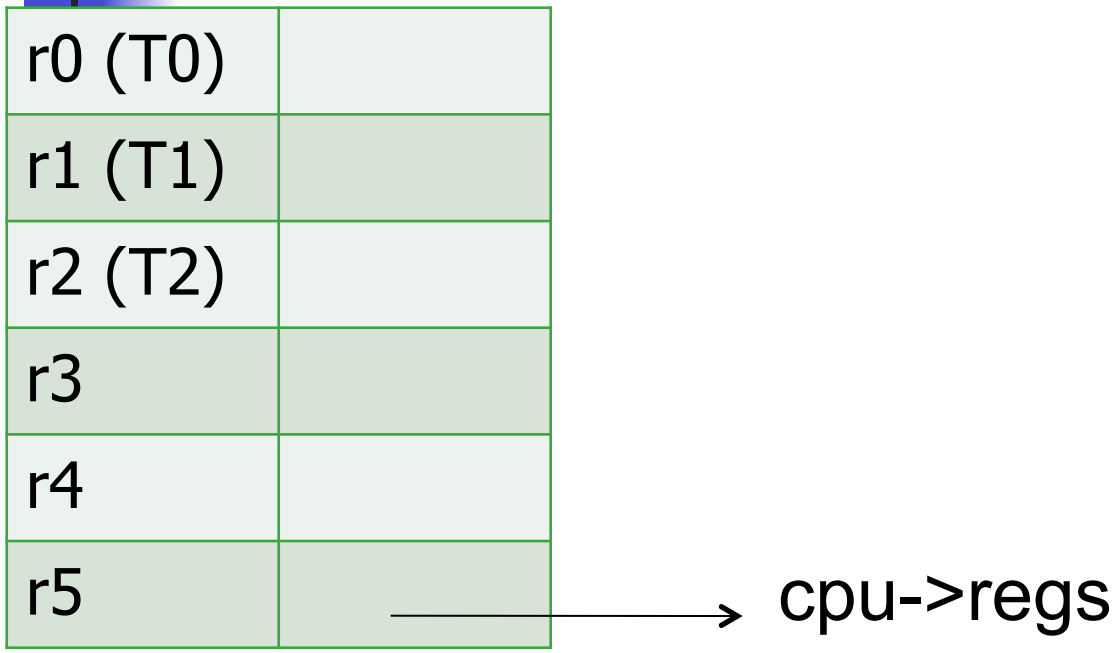
r0 (T0)	
r1 (T1)	
r2 (T2)	
r3	
r4	
r5	

cpu->regs

Physical Registers

Why don't we use one-to-mapping from guest regs to host regs?

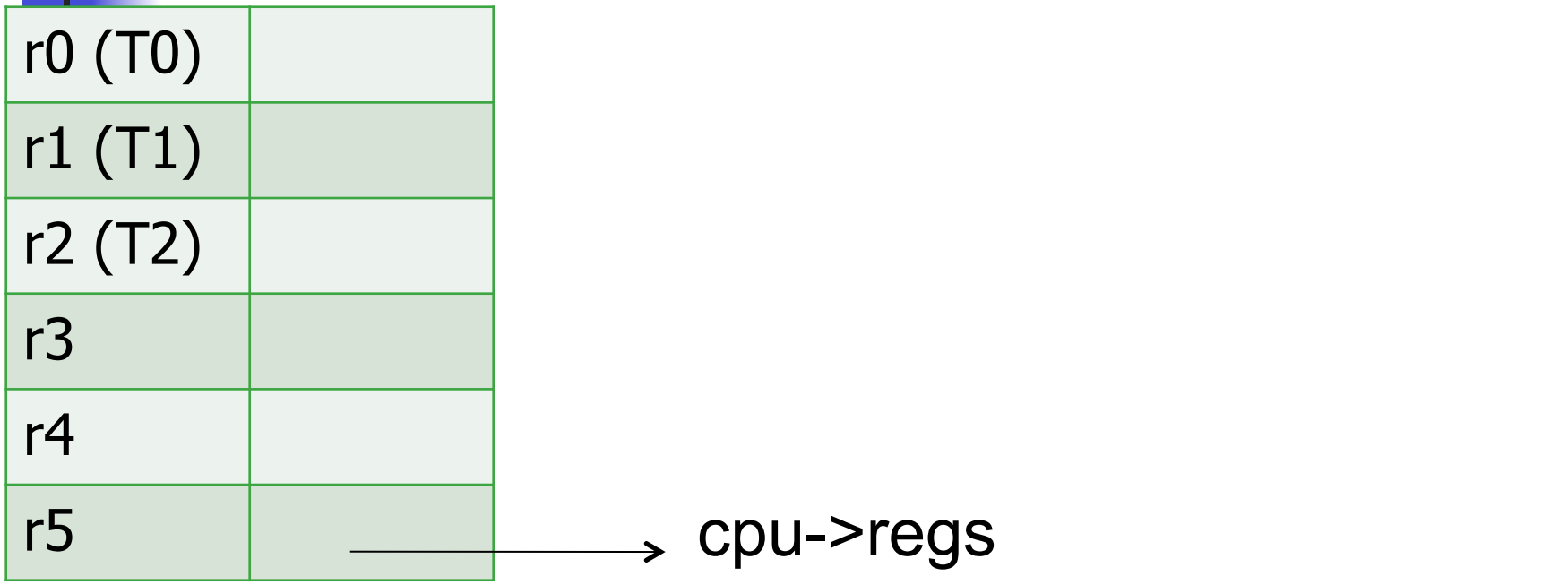
Dynamic Binary Translation



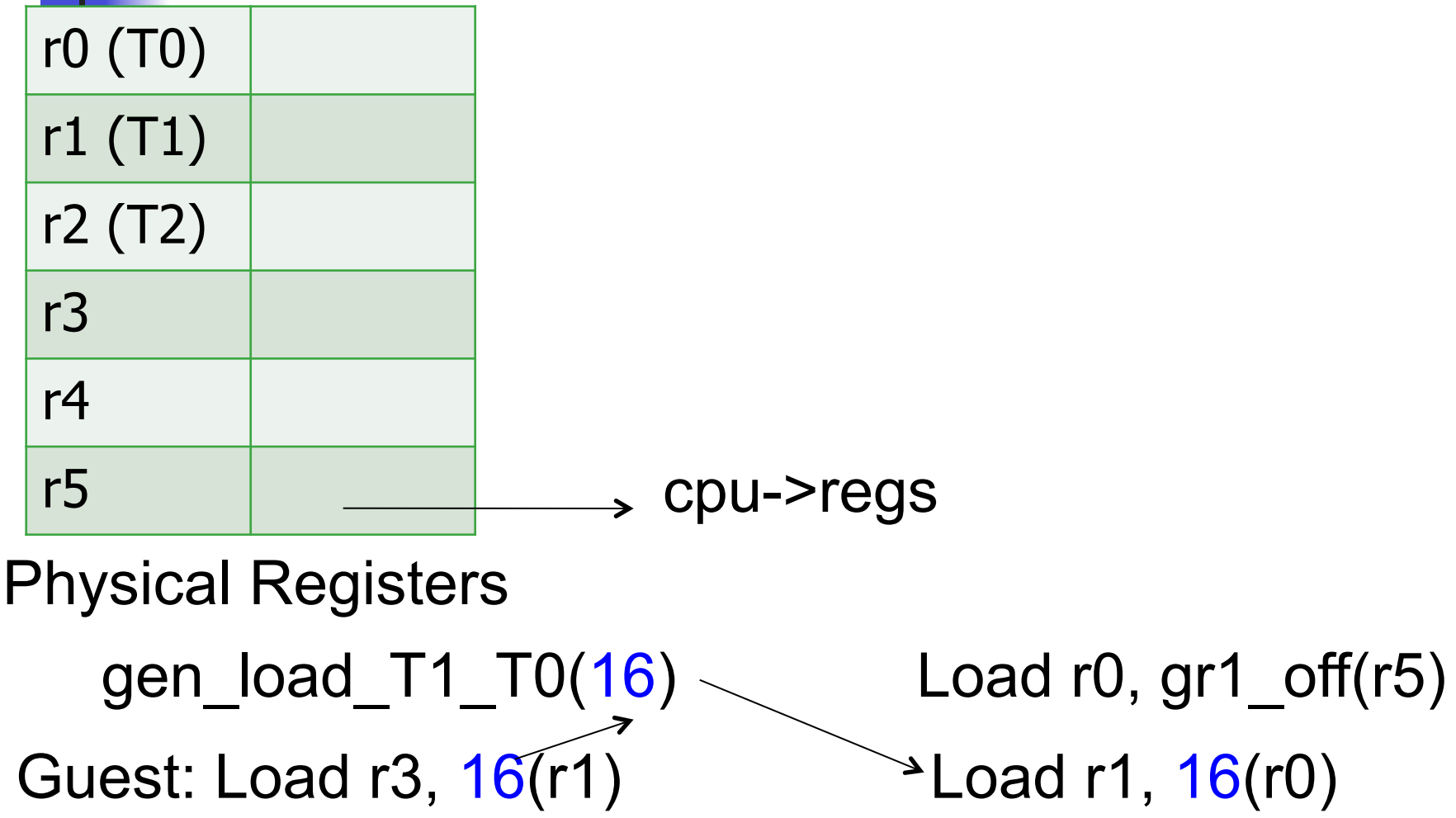
Guest: Load r3, 16(r1)

Load r1, imm(r0)

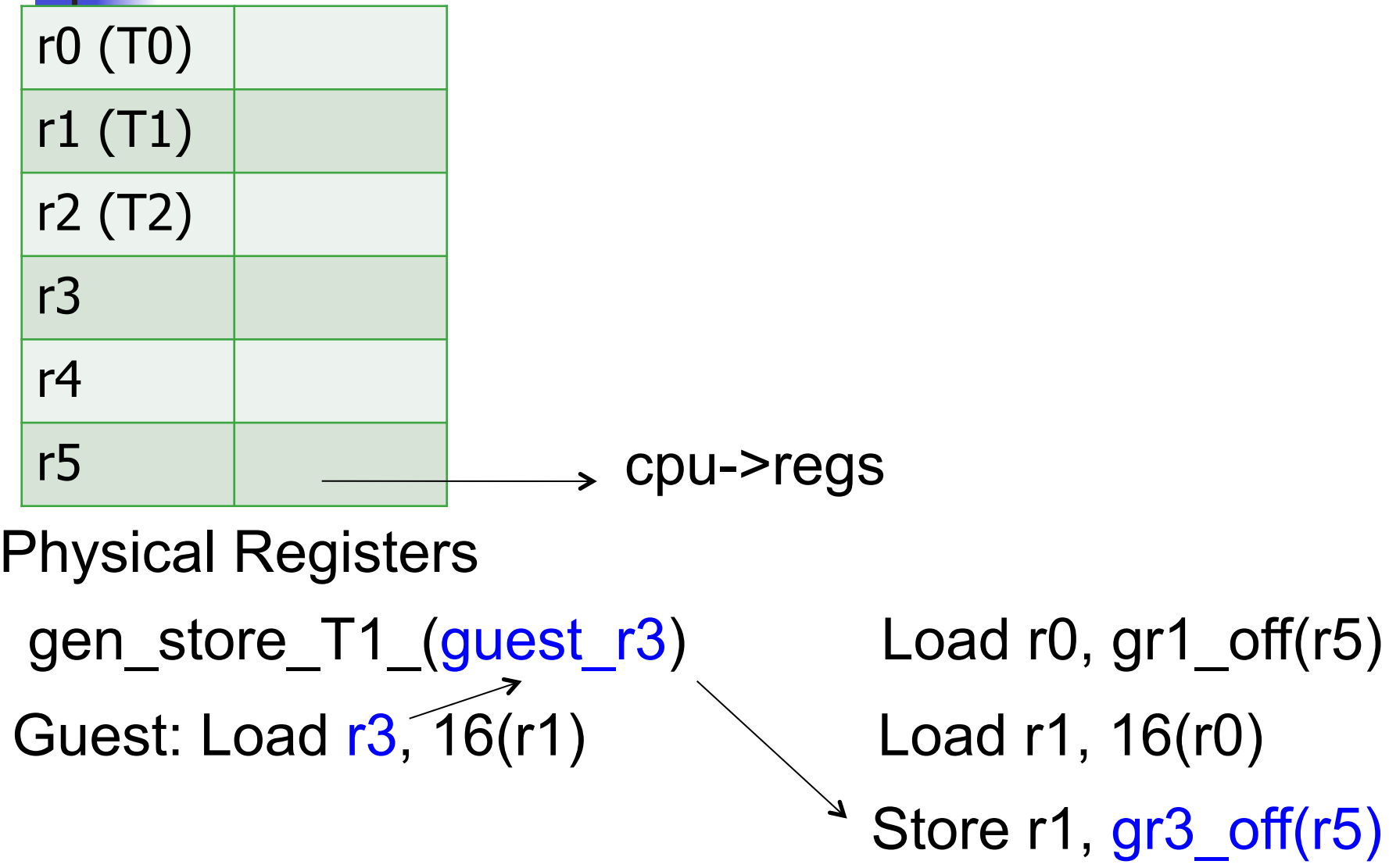
Dynamic Binary Translation



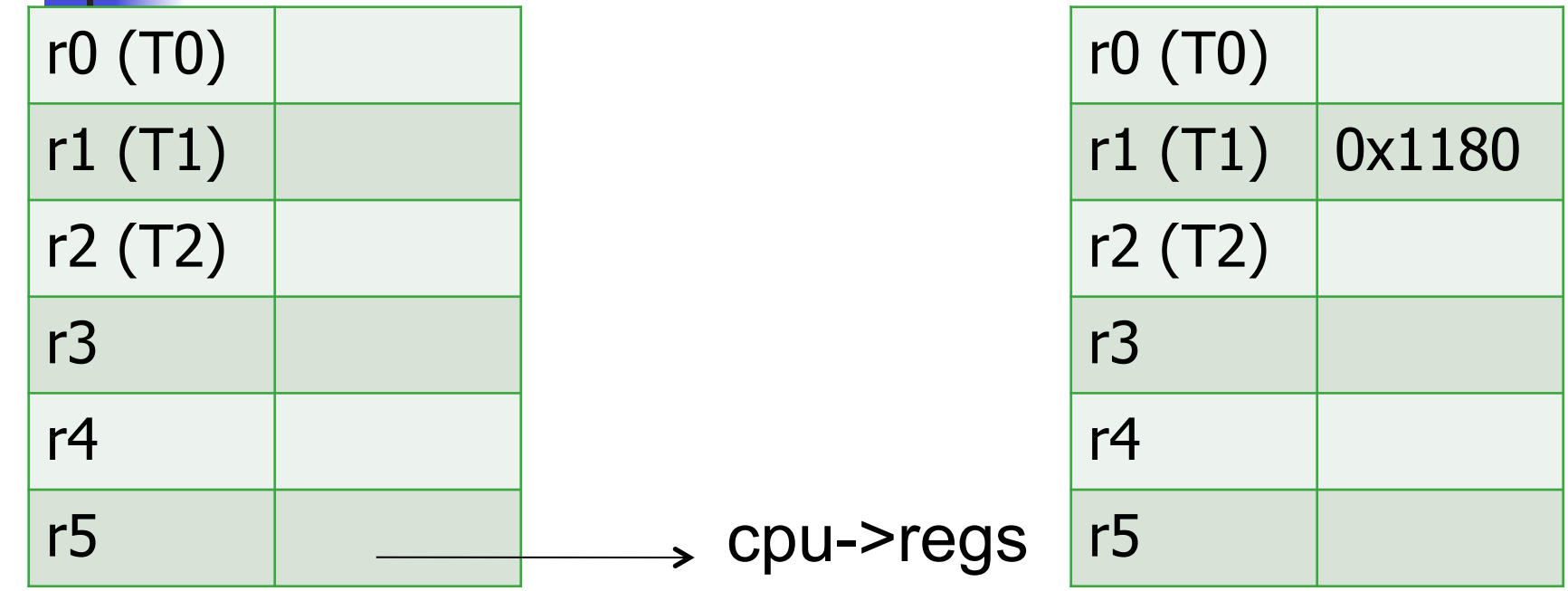
Dynamic Binary Translation



Dynamic Binary Translation



Dynamic Binary Translation



Load r0, gr1_off(r5)

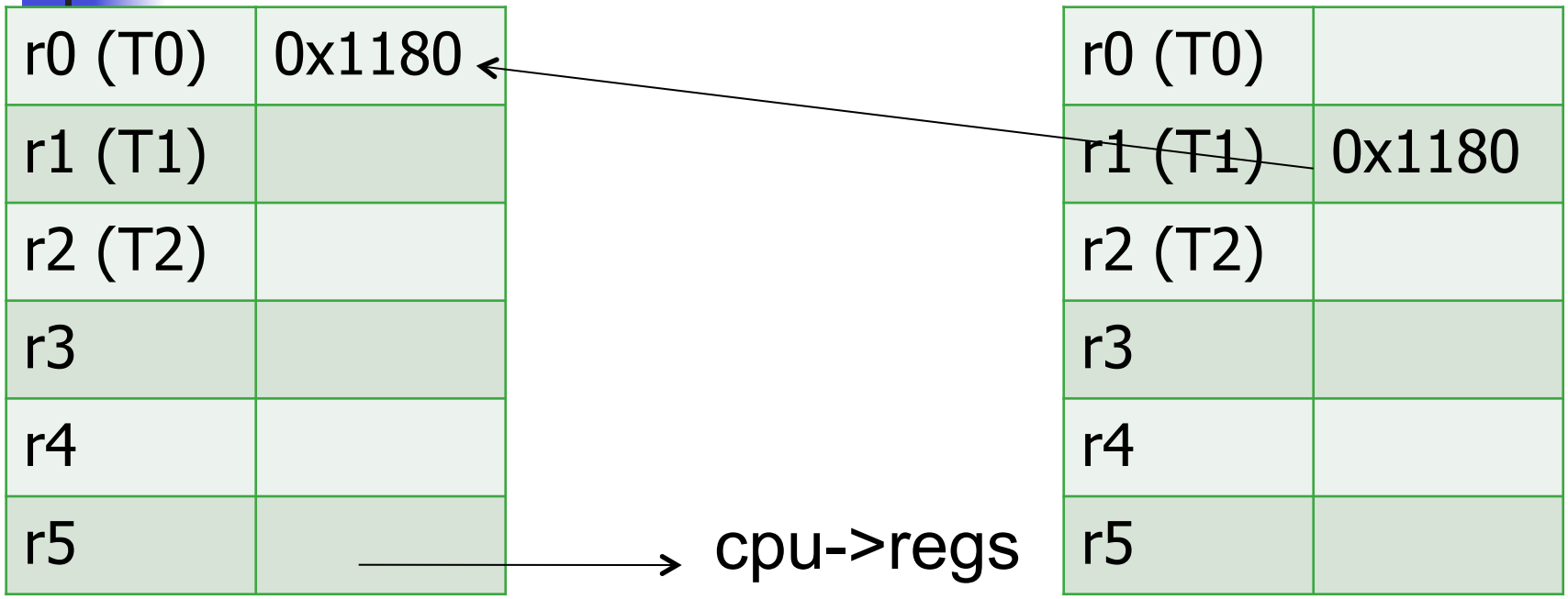
Guest: Load r3, 16(r1)

Load r1, 16(r0)

Guest mem: 0x1190 -> 0x1234

Store r1, gr3_off(r5)

Dynamic Binary Translation



Load r0, gr1_off(r5)

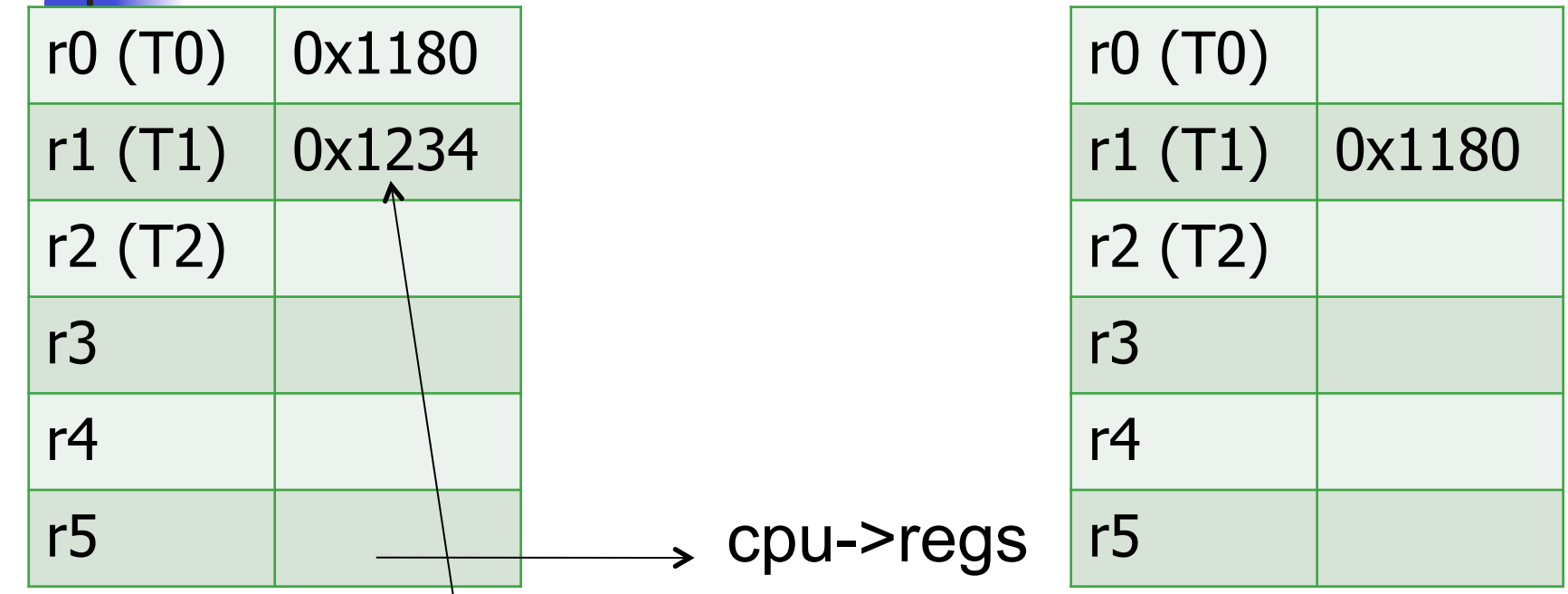
Guest: Load r3, 16(r1)

Guest mem: 0x1190 -> 0x1234

Load r1, 16(r0)

Store r1, gr3_off(r5)

Dynamic Binary Translation



Load r0, gr1_off(r5)

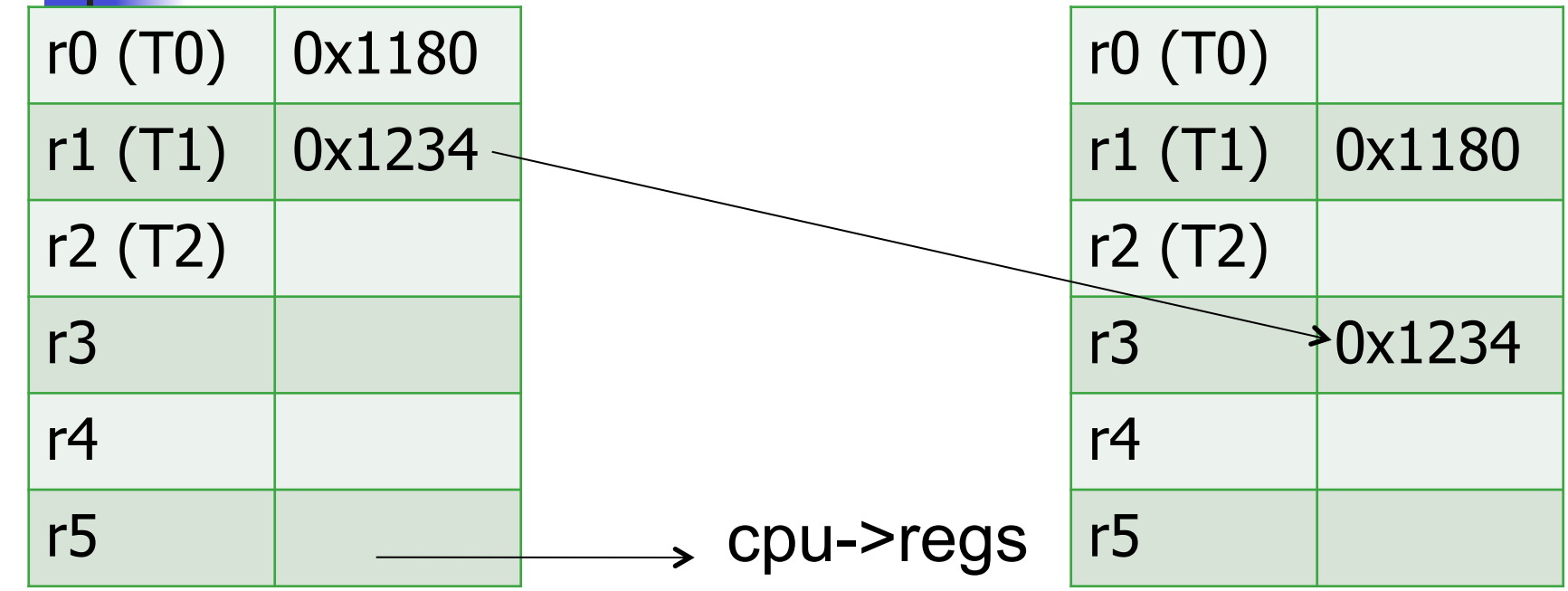
Guest: Load r3, 16(r1)

Load r1, 16(r0)

Guest mem: 0x1190 -> 0x1234

Store r1, gr3_off(r5)

Dynamic Binary Translation



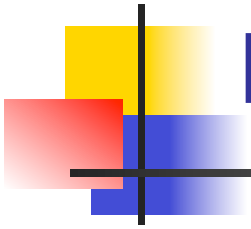
Load r0, gr1_off(r5)

Guest: Load r3, 16(r1)

Load r1, 16(r0)

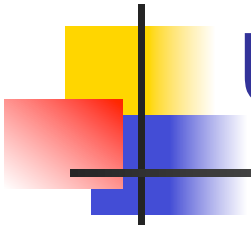
Guest mem: 0x1190 -> 0x1234

Store r1, gr3_off(r5)



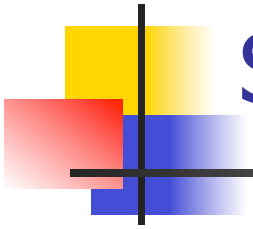
Exercise

- Generate the translation code for bitwise OR of guest code
- E.g., Or r4, r2, r1 -> r4 = r2 | r1
- Write any prototypes for code gen
  - E.g., gen_load_T1_T0(imm)
- Write the translation code
- Note: feel free to reuse any codegen from example

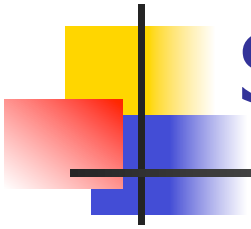


Useful functions

- Instruction: or r4, r2, r1
  - r4 = r2 | r1
- Gen_load_T0(guest reg no)
- Gen_load_T1(guest reg no)
- Gen_store_T2(guest reg no)



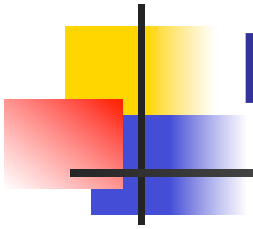
Solution



Solution

- or r4, r2, r1

- Gen_load_T0(gr2_off)
- Gen_load_T1(gr1_off)
- Gen_or_T2_T0_T1()
  - Or r2, r0, r1
- Gen_store_T2(gr4_off)

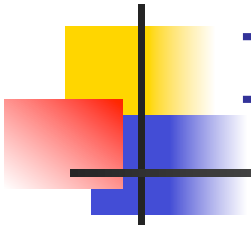


Dynamic binary translation for x86

- 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

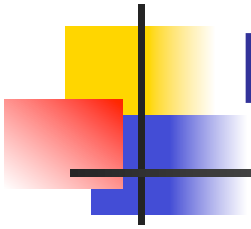


Inst-by-inst simulation: fetch

```
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
```

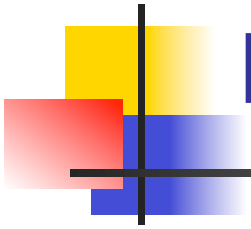
Inst-by-inst simulation: decode

```
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

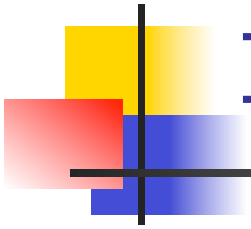


Dynamic translation for x86

- Map guest registers into these host registers

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

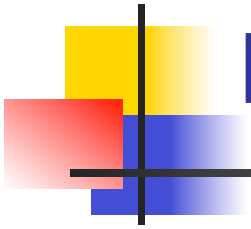
- Rdi -> holds a pointer to cpu



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

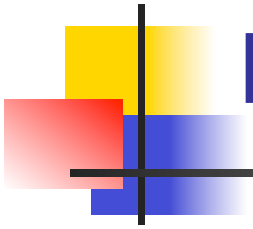
```
cpu->reg[control->rm] =
      cpu->reg[control->reg];
```

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



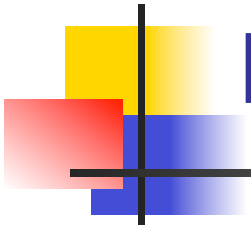
DBT: Mov %rsp, %rbp

```
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));
}
```



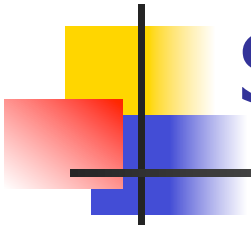
DBT: Mov %rsp, %rbp

```
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

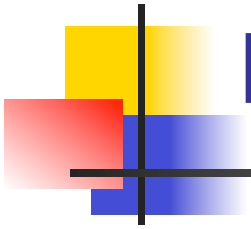
```
// Mov %rsp, %rbp
gen_load_T0(control->reg)
gen_store_T0(control->rm)

//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?



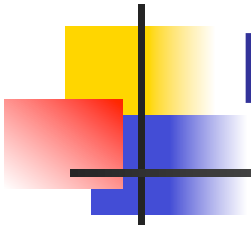
DBT optimizations

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