

CS152: Computer Systems Architecture

Some ISA Classifications



Sang-Woo Jun

Winter 2022

Eight great ideas

- Design for Moore's Law
- Use abstraction to simplify design
- Make the common case fast
- Performance via parallelism
- Performance via pipelining
- Performance via prediction
- Hierarchy of memories
- Dependability via redundancy

today



The RISC/CISC Classification

□ Reduced Instruction-Set Computer (RISC)

- Precise definition is debated
- Small number of more general instructions
 - RISC-V base instruction set has only dozens of instructions
 - **Memory load/stores not mixed with computation operations**
(Different instructions for load from memory, perform computation in register)
 - Often fixed-width encoding (4 bytes for base RISC-V)
- Complex operations implemented by composing general ones
 - Compilers try their best!
- RISC-V, ARM (Advanced RISC Machines),
MIPS (Microprocessor without Interlocked Pipelined Stages),
SPARC, ...

The RISC/CISC Classification

❑ Complex Instruction-Set Computer (CISC)

- Precise definition is debated (Not RISC?)
- Many, complex instructions
 - Various memory access modes per instruction (load from memory? register? etc)
 - Typically variable-length encoding per instruction
 - Modern x86 has thousands!
- Intel x86,
IBM z/Architecture,
- ...

The RISC/CISC Classification

- ❑ RISC paradigm is winning out
 - Simpler design allows faster clock
 - Simpler design allows efficient microarchitectural techniques
 - Superscalar, Out-of-order, ...
 - Compilers very good at optimizing software
- ❑ Most modern CISC processors have RISC internals
 - CISC instructions translated on-the-fly to RISC by the front-end hardware
 - Added overhead from translation (silicon, power, performance, ...)

CS152: Computer Systems Architecture

RISC-V Introduction



Sang-Woo Jun

Winter 2022

Course outline

- ❑ Part 1: The Hardware-Software Interface
 - What makes a 'good' processor?
 - Assembly language and programming conventions
- ❑ Part 2: Recap of digital design
 - Combinational and sequential circuits
 - How their restrictions influence processor design
- ❑ Part 3: Computer Architecture
 - Computer Arithmetic
 - Simple and pipelined processors
 - Caches and the memory hierarchy
- ❑ Part 4: Computer Systems
 - Operating systems, Virtual memory

Why learn assembly?

- ❑ We (typically) don't program with assembly any more
- ❑ BUT, important to understand architecture
 - Arithmetic in x86 has two operands (e.g., add eax ebx), while RISC-V has three (e.g., add x5 x6 x7)
 - x86 has six general-purpose registers, while RISC-V has 32
 - What drove these decisions? How does this impact processor design and performance?

We need a reference architecture

RISC-V Introduction

- ❑ We use RISC-V as a learning tool
- ❑ A free and open ISA from Berkeley
 - A clean-slate design using what was learned over decades
 - Uncluttered by backwards compatibility
 - Simplicity-oriented (Some say to a fault!)
- ❑ Many, many industry backers!
 - Google, Qualcomm, NVIDIA, IBM, Samsung, Huawei, ...



RISC-V Introduction

❑ Composable, modular design

- Consists of a base ISA -- RV32I (32 bit), RV64I (64 bit) We will use RV32I
- And many composable extensions. Including:
 - 'M': Math extension. Multiply and divide
 - 'F', 'D': Floating point extensions, single and double precision
 - 'A': Atomic operations
 - 'B': Bit manipulation
 - 'T': Transactional memory
 - 'P': Packed SIMD (Single-Instruction Multiple Data)
 - 'V': Vector operators
 - Designer can choose to implement combinations: e.g., RV64IMFT

❑ Virtual memory (Sv32, Sv48) and privileged operations specified

Structure of the ISA

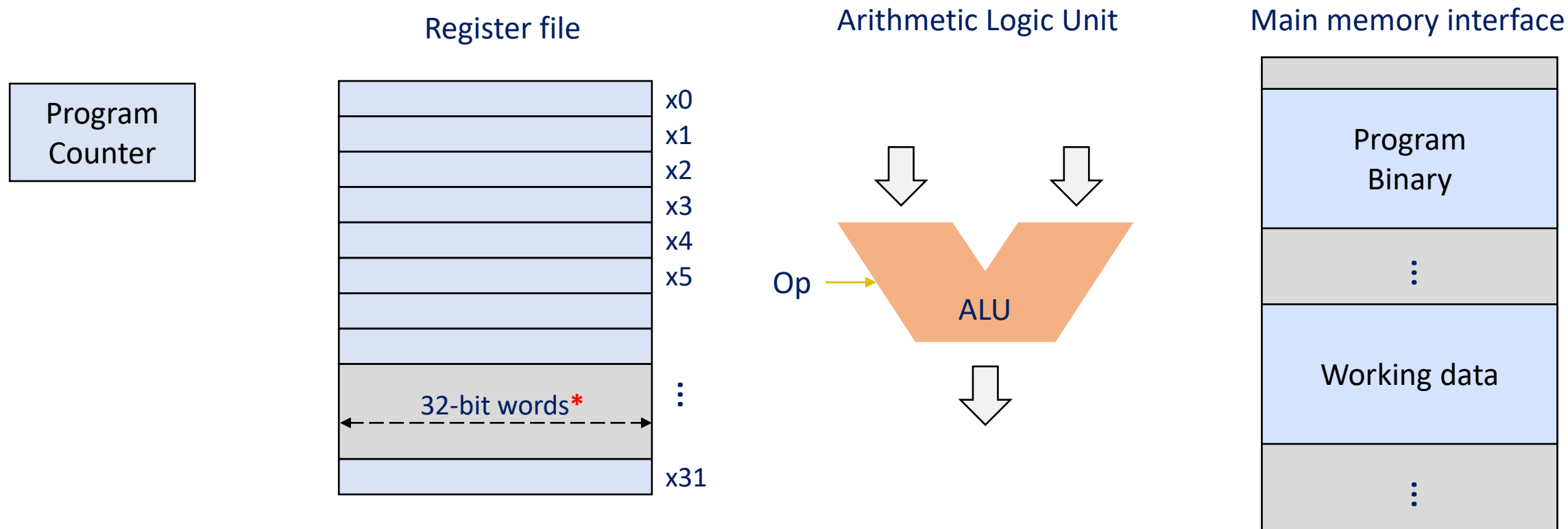
❑ Small amount of fixed-size registers

- For RV32I, 32 32-bit registers (32 64-bit registers for RV64)
- A question: **Why isn't this number larger?** Why not 1024 registers?
- Another question: Why not zero?

❑ Three types of instructions

1. Computational operation: from register file to register file
 - $x_d = \text{Op}(x_a, x_b)$, where $\text{Op} \in \{+, -, \text{AND}, \text{OR}, >, <, \dots\}$
 - Op implemented in ALU
2. Load/Store: between memory and register file
3. Control flow: jump to different part of code

RISC-V base architecture components



- Current location in program execution

- 32 32-bit registers
- (64 bit words for RV64)

- Input: 2 values, Op
- Output: 1 value

Op \in {+, -, AND, OR, >, <, ...}

- Actual memory outside CPU chip

Super simplified processor operation

```
inst = mem[PC]
```

```
next_PC = PC + 4
```

```
if ( inst.type == STORE ) mem[rf[inst.arg1]] = rf[inst.arg2]
```

```
if ( inst.type == LOAD ) rf[inst.arg1] = mem[rf[inst.arg2]]
```

```
if ( inst.type == ALU ) rf[inst.arg1] = alu(inst.op, rf[inst.arg2], rf[inst.arg3])
```

```
if ( inst.type == COND ) next_PC = rf[inst.arg1]
```

```
PC = next_PC
```

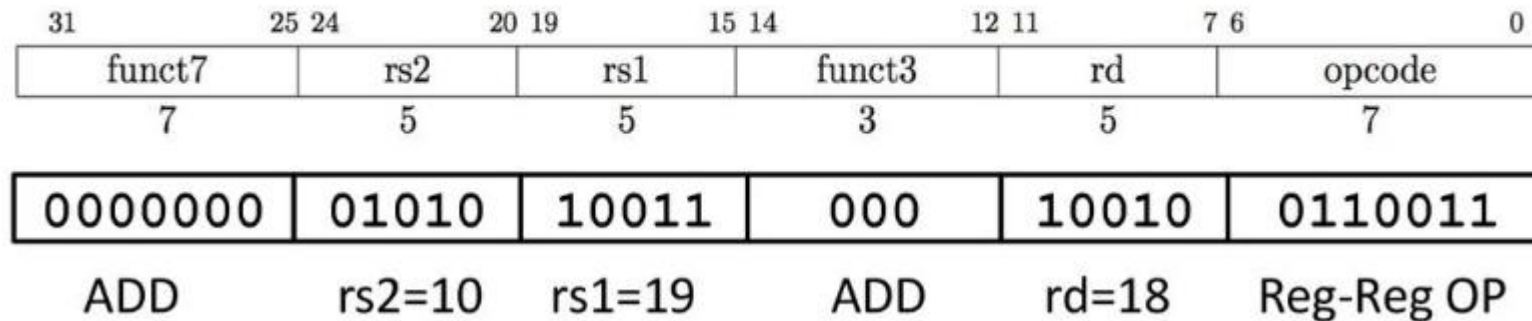
RISC-V never mixes memory and ALU operations!

In the four bytes of the instruction,
type, arg1, arg2, arg3, op
needs to be encoded

A RISC-V Example (“00A9 8933”)

- This four-byte binary value will instruct a RISC-V CPU to perform
 - add values in registers x19 x10, and store it in x18
 - regardless of processor speed, internal implementation, or chip designer

`add x18,x19,x10`



In the four bytes of the instruction, **type, arg1, arg2, arg3, op** needs to be encoded

Aside: CISC and x86

- x86 ISA is CISC (“Complex”)

Hex	Mnemonics
C3	ret
48 b8 88 77 66 55 44 33 22 11	movabs rax,0x1122334455667788
64 ff 03	DWORD PTR fs:[ebx]
64 67 66 f0 ff 07	lock inc WORD PTR fs:[bx]
2e c4 e2 71 96 84 be 34 23 12 01	vfmaddsub132ps xmm0, xmm1, xmmword ptr cs: [esi + edi * 4 + 0x11223344]

CS152: Computer Systems Architecture

RISC-V Assembly



Sang-Woo Jun

Winter 2022



Large amount of material adapted from MIT 6.004, “Computation Structures”,
Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”,
and CS 152 Slides by Isaac Scherson

Three types of instructions

1. Computational operation: from register file to register file
2. Load/Store: between memory and register file
3. Control flow: jump to different part of code

Computational operations

- ❑ Arithmetic, comparison, logical, shift operations
- ❑ Register-register instructions
 - 2 source operand registers
 - 1 destination register
 - Format: op dst, src1, src2

Arithmetic	Comparison	Logical	Shift
add, sub	slt, sltu	and, or, xor	sll, srl, sra

set less than
set less than unsigned

Signed/unsigned?

Shift left logical
Shift right logical
Shift right arithmetic

Arithmetic/logical?

Computational operations

□ Register-immediate operations

- 2 source operands
 - One register read
 - One immediate value encoded in the instruction **Limited to 12 bits! (Why?)**
- 1 destination register
- Format: op dst, src, imm
 - eg., addi x1, x2, 10

Format	Arithmetic	Comparison	Logical	Shift
register-register	add, sub	slt, sltu	and, or, xor	sll, srl, sra
register-immediate	addi	slti, sltiu	andi, ori, xori	slli, srli, srai

No “subi” instead use negative with “addi”

Aside: Signed and unsigned operations

- ❑ Registers store 32-bits of data, no type
- ❑ Some operations interpret data as signed, some as unsigned values

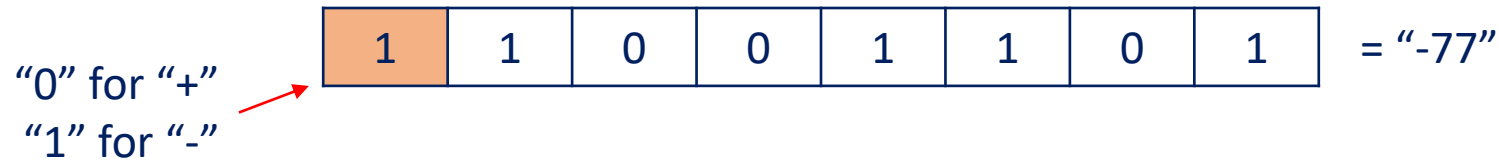
operation	Meaning
add d, a, b	$d = sx(a) + sx(b)$
slt d, a, b	$d = sx(a) > sx(b) ? 1 : 0$
sltu d, a, b	$d = ux(a) > ux(b) ? 1 : 0$
sll d, a, b	$d = ux(a) \ll b$
srl d, a, b	$d = ux(a) \gg b$
sra d, a, b	$d = sx(a) \gg b$

sx: interpret as signed, ux, interpret as unsigned

No sla operation. Why? Two's complement ensures sla == sll

Aside: Two's complement encoding

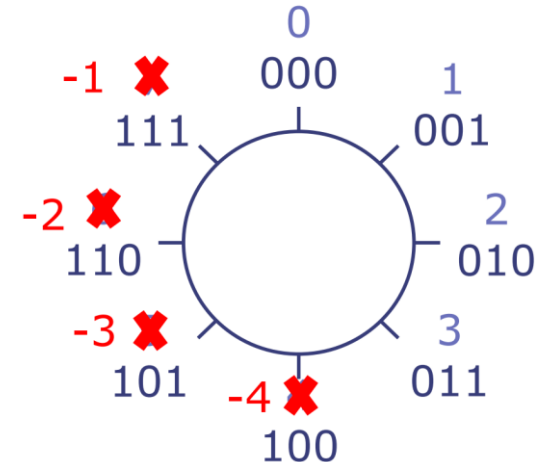
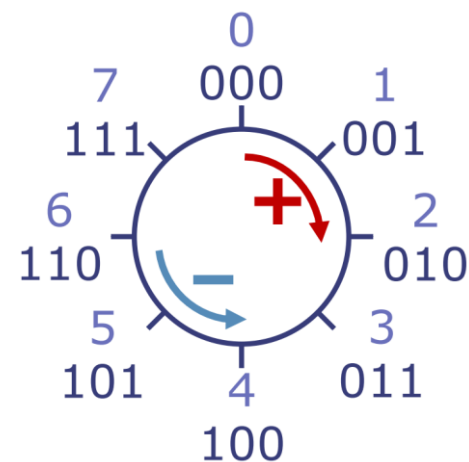
- ❑ How should we encode negative numbers?
- ❑ Simplest idea: Use one bit to store the sign



- ❑ Is this a good encoding? No!
 - Two representations for "0" ("+0", "-0")
 - Add and subtract require different algorithms

Aside: Two's complement encoding

- ❑ The larger half of the numbers are simply interpreted as negative
- ❑ Background: Overflow on fixed-width unsigned numbers wrap around
 - Assuming 3 bits, $100 + 101 = 1001$ (overflow!) = stores 001
 - “Modular arithmetic”, equivalent to following modN to all operations
- ❑ Relabeling allows natural negative operations via modular arithmetic
 - e.g., $111 + 010 = 1001$ (overflow!) = stores 001
equivalent to $-1 + 2 = 1$
 - Subtraction uses same algorithm as add
e.g., $a - b = a + (-b)$



Aside: Two's complement encoding

□ Some characteristics of two's encoded numbers

- Negative numbers have "1" at most significant bit (sign bit)
- Most negative number = $10\dots000 = -2^{N-1}$
- Most positive number = $01\dots111 = 2^{N-1} - 1$
- If all bits are 1 = $11\dots111 = -1$
- Negation works by flipping all bits and adding 1

$$-A + A = 0$$

$$-A + A = -1 + 1$$

$$-A = (-1 - A) + 1$$

$$-A = \sim A + 1$$

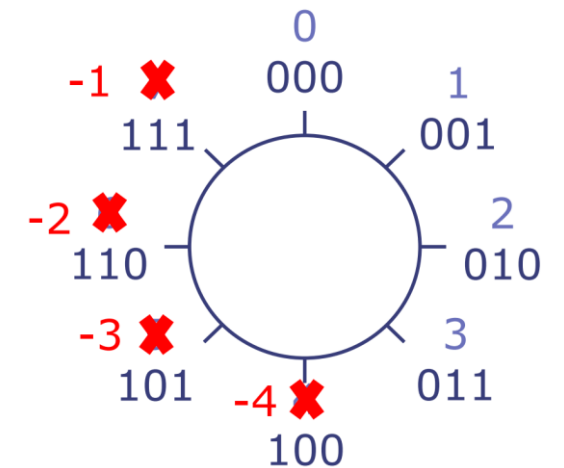
Because -1 is all 1s,
there is no borrowing,
therefore subtracting A from -1 is flipping all bits

e.g.,

111111

-100010

=011101



Return to shifting with two's complement

□ Right shift requires both logical and arithmetic modes

- Assuming 4 bits
- $(4_{10}) \gg 1 = (0100_2) \gg 1 = 0010_2 = 2_{10}$ Correct!
- $(-4_{10}) \gg_{\text{logical}} 1 = (1100_2) \gg_{\text{logical}} 1 = 0110_2 = 6_{10}$ For signed values, Wrong!
- $(-4_{10}) \gg_{\text{arithmetic}} 1 = (1100_2) \gg_{\text{arithmetic}} 1 = 1110_2 = -2_{10}$ Correct!
- Arithmetic shift replicates sign bits at MSB

□ Left shift is the same for logical and arithmetic

- Assuming 4 bits
- $(2_{10}) \ll 1 = (0010_2) \ll 1 = 0100_2 = 4_{10}$ Correct!
- $(-2_{10}) \ll_{\text{logical}} 1 = (1110_2) \ll_{\text{logical}} 1 = 1100_2 = -4_{10}$ Correct!

Three types of instructions

1. Computational operation: from register file to register file
2. Load/Store: between memory and register file
3. Control flow: jump to different part of code

Load/Store operations

❑ Format: op dst, offset(base)

- Address specified by a pair of <base address, offset>
- e.g., lw x1, 4(x2) # Load a word (4 bytes) from [x2]+4 to x1
- The offset is a small constant

❑ Variants for types

- lw/sw: Word (4 bytes)
- lh/lhu/sh: Half (2 bytes)
- lb/lbu/sb: Byte (1 byte)
- 'u' variant is for unsigned loads
 - Half and Byte reads extends read data to 32 bits. Signed loads are sign-bit aware

Sign extension

- ❑ Representing a number using more bits
 - Preserve the numeric value
- ❑ Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- ❑ Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110
- ❑ In RISC-V instruction set
 - `lb`: sign-extend loaded byte
 - `lbu`: zero-extend loaded byte

Why doesn't stores need sign variants?

Three types of instructions

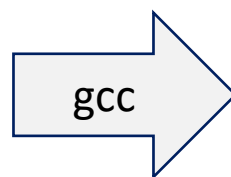
1. Computational operation: from register file to register file
2. Load/Store: between memory and register file
3. Control flow: jump to different part of code

Control flow instructions - Branching

- ❑ Format: cond src1, src2, label
- ❑ If condition is met, jump to label. Otherwise, continue to next

beq	bne	blt	bge	bltu	bgeu
==	!=	<	>=	<	>=

```
if (a < b):    c = a + 1
else:         c = b + 2
```



```
        bge x1, x2, else
        addi x3, x1, 1
        beq x0, x0, end
else:   addi x3, x2, 2
end:
```

(Assume x1=a; x2=b; x3=c;)

Control flow instructions – Jump and Link

□ Format:

- jal dst, label – Jump to 'label', store PC+4 in dst
- jalr dst, offset(base) – Jump to rf[base]+offset, store PC+4 in dst
 - e.g., jalr x1, 4(x5) Jumps to x5+4, stores PC+4 in x1

□ Why do we need two variants?

- jal has a limit on how far it can jump
 - (Why? Encoding issues explained later)
- jalr used to jump to locations defined at runtime
 - Needed for many things including function calls (e.g., Many callers calling one function)

```
...
jal x1, function1
...
function1:
...
jalr x0, 0(x1)
```

Three types of instructions – Part 4

1. Computational operation: from register file to register file
2. Load/Store: between memory and register file
3. Control flow: jump to different part of code
4. Load upper immediate: Load (relatively) large immediate value

Problem: Addi can load up to 12 bits! How do we encode large values?

Load upper immediate instructions

❑ LUI: Load upper immediate

- `lui dst, immediate` → $dst = immediate \ll 12$
- Can load $(32-12 = 20)$ bits
- Used to load large (~ 32 bits) immediate values to registers
- `lui` followed by `addi` (load 12 bits) to load 32 bits

❑ AUIPC: Add upper immediate to PC

- `auipc, dst, immediate` → $dst = PC + immediate \ll 12$
- Can load $(32-12 = 20)$ bits
- `auipc` followed by `addi`, then `jalr` to allow long jumps within any 32 bit address

Typically not used by human programmers!
Assemblers use them to implement complex operations

Aside: Notably missing: Condition codes

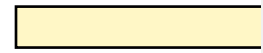
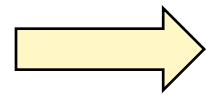
- ❑ Implicitly managed bitmap of flags
 - e.g., Carry, Overflow, Negative, Equal to zero, less than, ...
 - Flags set by previously executed instruction
- ❑ Some instructions can execute only if conditions are met
 - “Predicated instructions”
 - ARM MOVHS (Move higher or same) only moves if previous instruction resulted in “higher or same” flag being set. Otherwise NOP
 - Can remove a costly conditional branch instruction if used well
 - Carry bits can be useful for large adds, ...

Aside: Notably missing: Condition codes

□ Predicated instructions in ARM

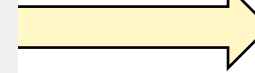
```
if (a > 10) {  
    a = 10;  
} else {  
    a = a + 1;  
}
```

C Code



```
    cmp     r0, #10  
    blo     r0_is_small  
r0_is_big:  
    mov     r0, #10  
    b      continue  
r0_is_small:  
    add     r0, r0, #1  
continue:  
    @ Other code.
```

Without predicated instructions



```
    cmp     r0, #10  
    movhs   r0, #10  
    addlo   r0, r0, #1
```

With predicated instructions

Aside: Notably missing: Condition codes

- ❑ RISC-V does not have this
 - Designers wanted simpler communications between pipeline stages

CS152: Computer Systems Architecture

RISC-V ISA Encoding



Sang-Woo Jun

Winter 2022



Large amount of material adapted from MIT 6.004, “Computation Structures”,
Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”,
and CS 152 Slides by Isaac Scherson

What does the ISA for this look like?

- ❑ ADD: 0x000000001,
SUB: 0x000000002,
LW: 0x000000003,
SW: 0x000000004, ...?
- ❑ Haphazard encoding makes processor design complicated!
 - More chip resources, more power consumption, less performance

RISC-V instruction encoding

❑ Restrictions

- 4 bytes per instruction
- Different instructions have different parameters (registers, immediates, ...)
- Various fields should be encoded to consistent locations
 - Simpler decoding circuitry

❑ Answer: RISC-V uses 6 “types” of instruction encoding

Name (Field Size)	Field						Comments
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

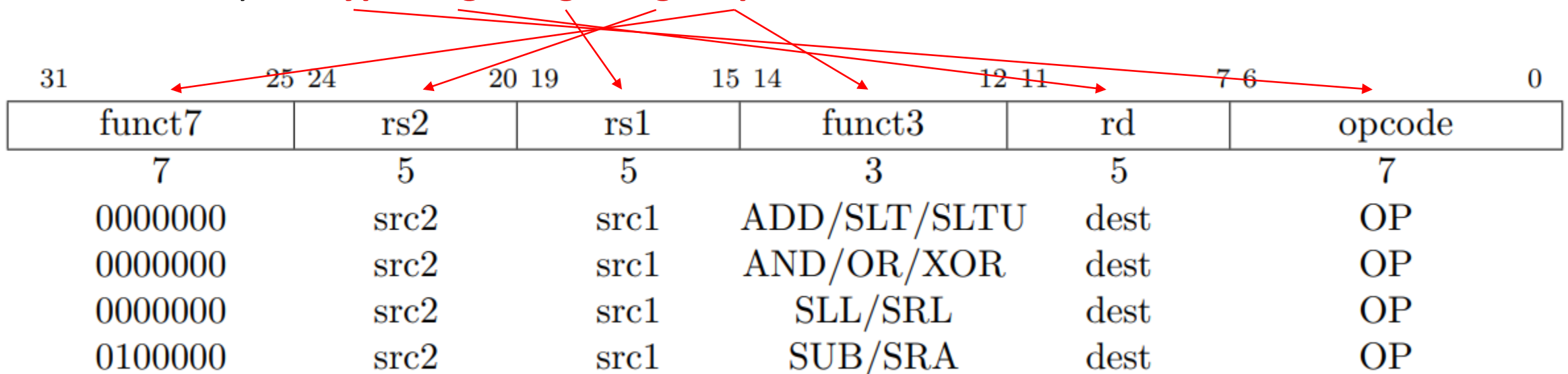
We're not going to look at everything...

R-Type encoding

❑ Relatively straightforward, register-register operations encoding

❑ Remember:

- if (inst.type == ALU) $rf[inst.arg1] = alu(inst.op, rf[inst.arg2], rf[inst.arg3])$
- In 4 bytes, **type**, **arg1**, **arg2**, **arg3**, **op** needs to be encoded



R-Type encoding

□ Instruction fields

- opcode: operation code
- rd: destination register number (5 bits for 32 registers)
- funct3: 3-bit function code (additional opcode)
- rs1: the first source register number (5 bits for 32 registers)
- rs2: the second source register number (5 bits for 32 registers)
- funct7: 7-bit function code (additional opcode, funct3 only support 8 functions)



R-Type encoding

□ Instruction fields

- opcode: operation code
- rd: destination register number (5 bits for 32 registers)
- funct3: 3-bit function code (additional opcode)
- rs1: the first source register number (5 bits for 32 registers)
- rs2: the second source register number (5 bits for 32 registers)
- funct7: 7-bit function code (additional opcode)

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

e.g., add x9,x20,x21

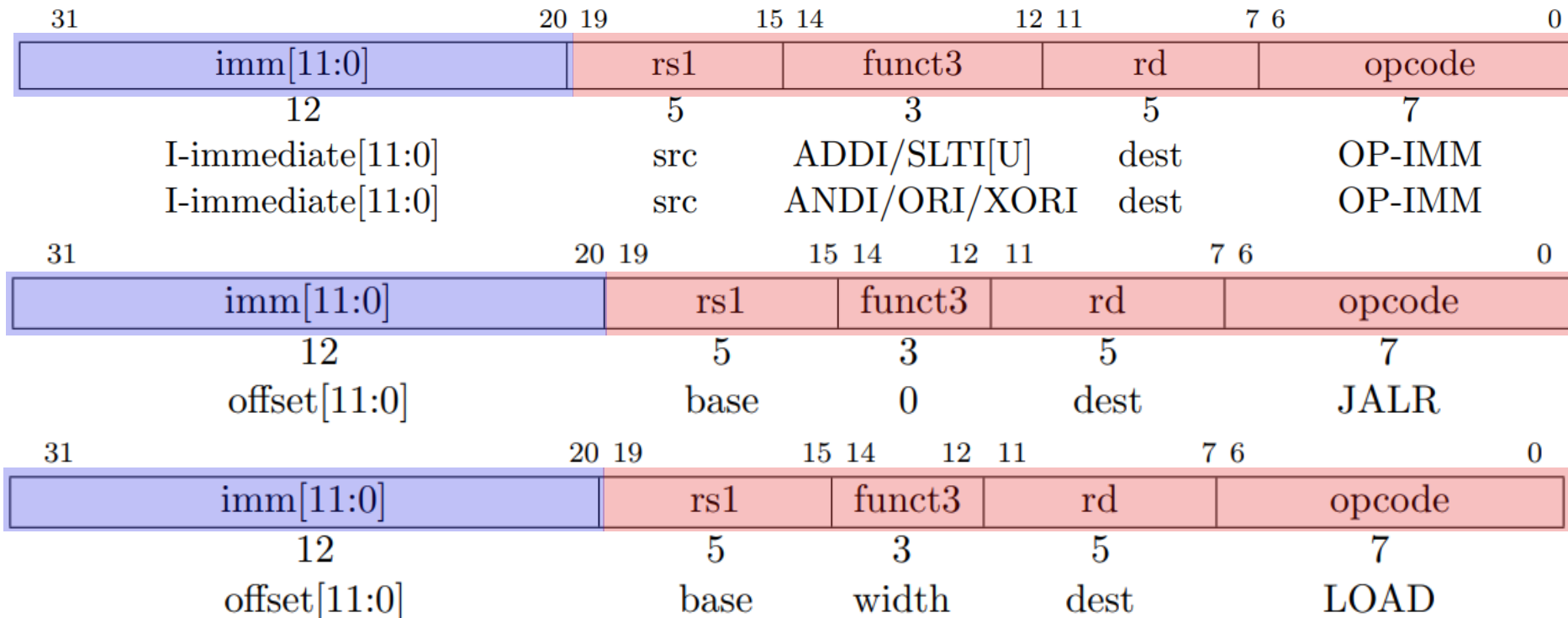
= 0x015A04B316

I-Type encoding

❑ Register-Immediate operations encoding

- One register, one immediate as input, one register as output

Operands in same location!

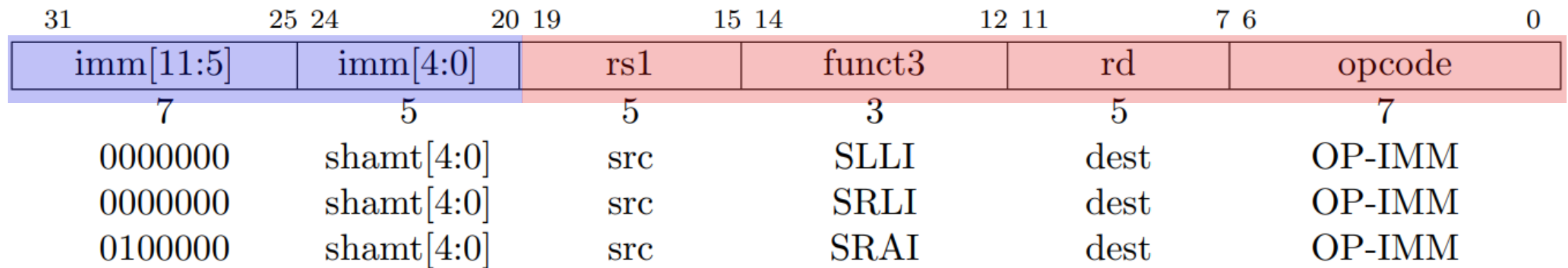


Immediate value limited to 12 bits signed!

addi x5, x6, 2048 # Error: illegal operands `addi x5,x6,2048'

I-Type encoding

- Shift instructions need only 5 bits for immediate (32 bit words)
 - Top 7 bits of the immediate field used as func7
 - I-Type func7 same location as R-type func7
 - Allows efficient reuse of decode circuitry

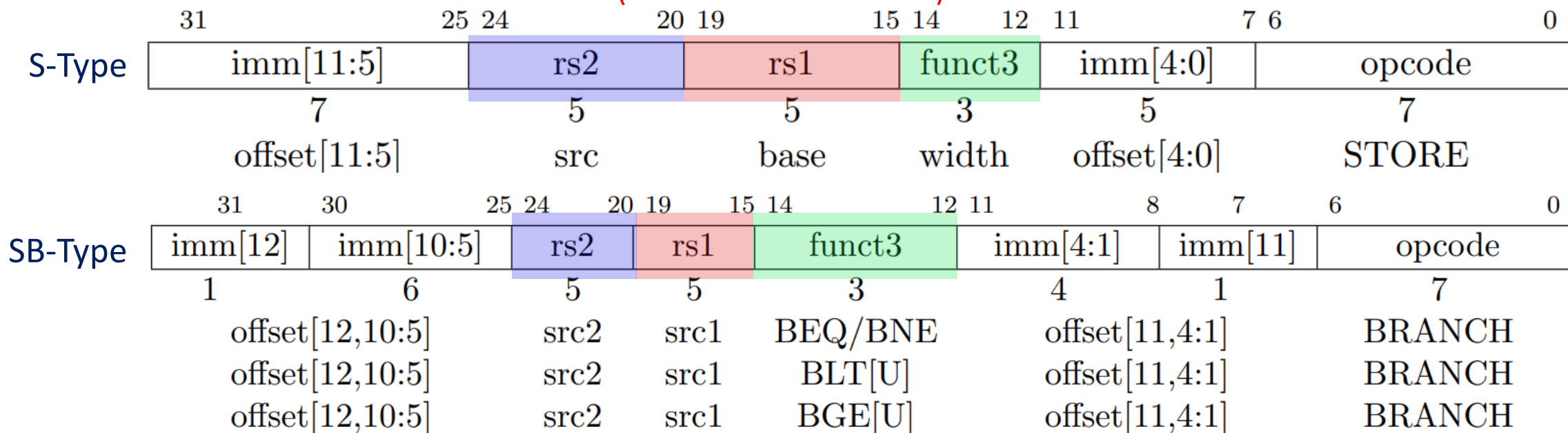


S-Type and SB-Type encoding

□ Store operation: two register input, no output

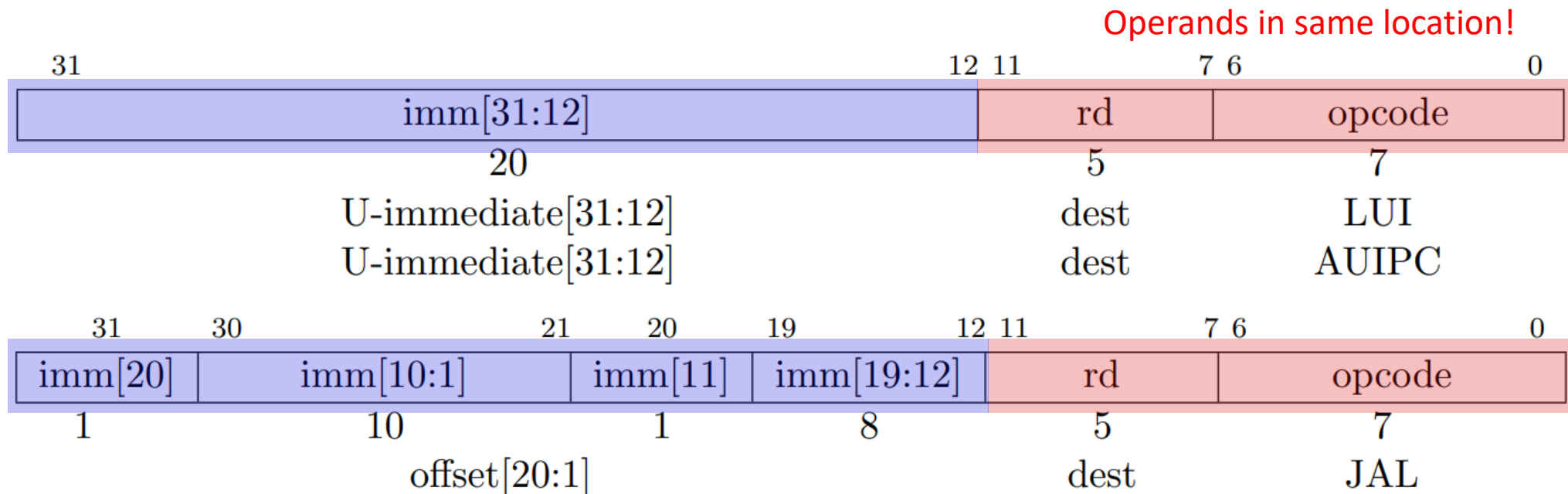
- e.g.,
sw src, offset(base)
beq r1, r2, label

Operands in same location!
(Bit width not to scale...)



U-Type and UJ-Type encoding

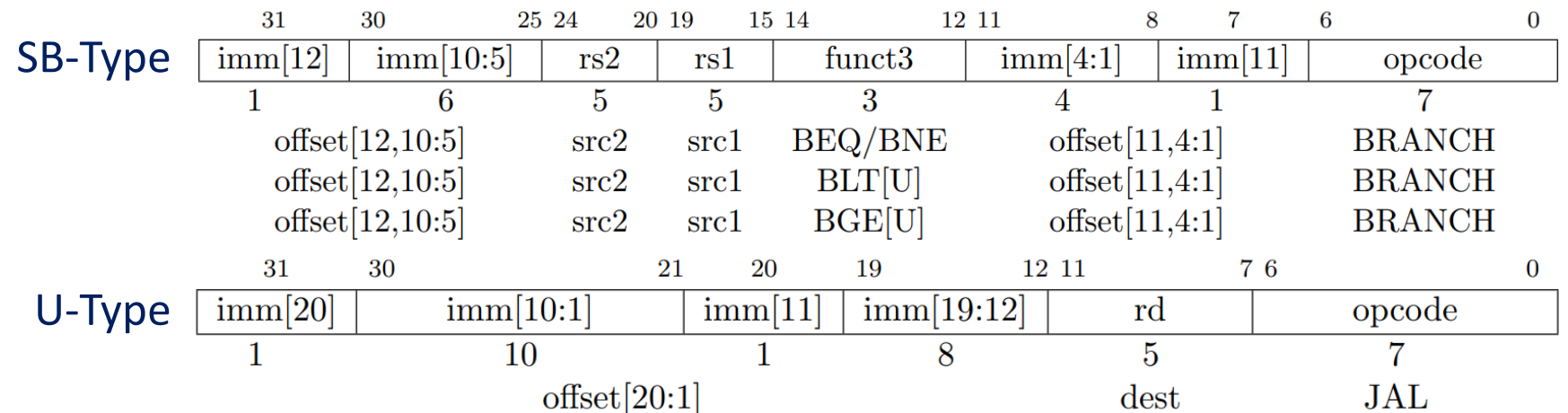
- ❑ One destination register, one immediate operand
 - U-Type: LUI (Load upper immediate), AUIPC (Add upper immediate to PC)
Typically not used by human programmer
 - UB-Type: JAL (Jump and link)



Relative addressing

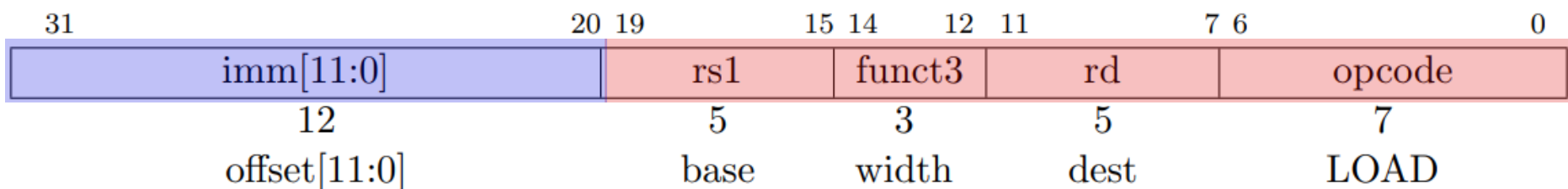
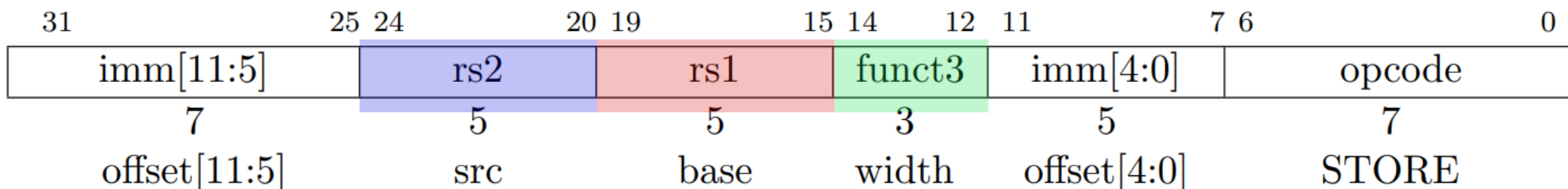
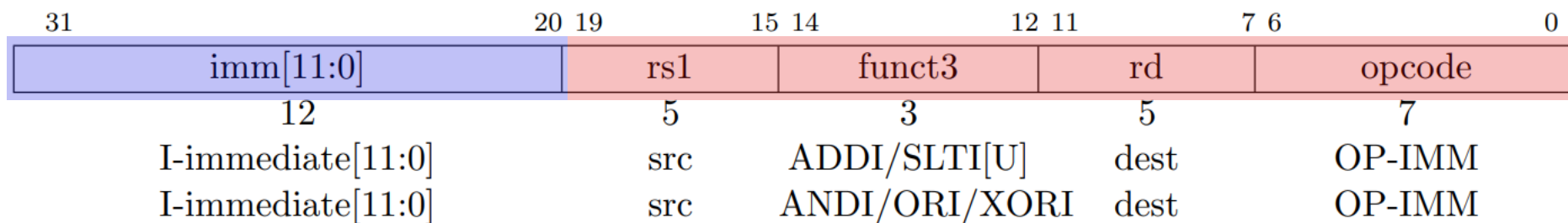
❑ Problem: jump target offset is small!

- For branches: 13 bits, For JAL: 21 bits
- How does it deal with larger program spaces?
- Solution: PC-relative addressing ($PC = PC + imm$)
 - Remember format: beq x5, x6, label
 - Translation from label to offset done by assembler
 - Works fine if branch target is nearby. If not, AUIPC and other tricks by assembler



Why is the immediate field 12 bits?

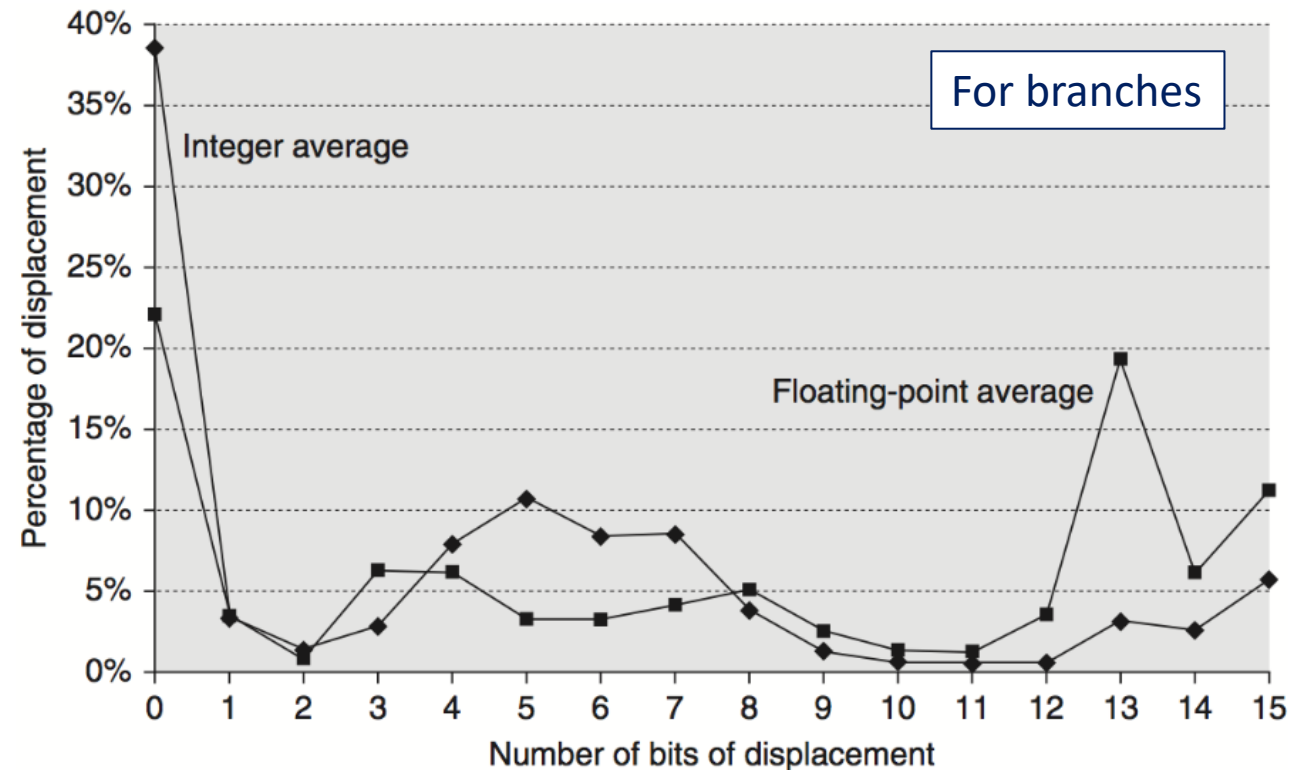
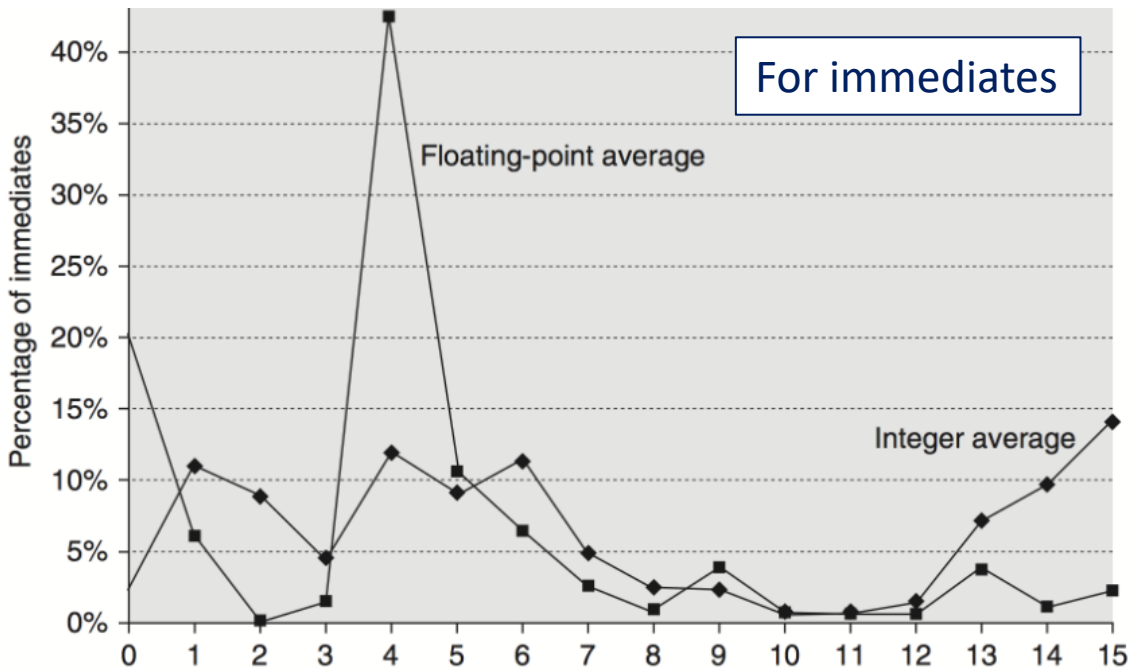
- If most immediate values are larger, this instruction is useless!



Benchmark-driven ISA design

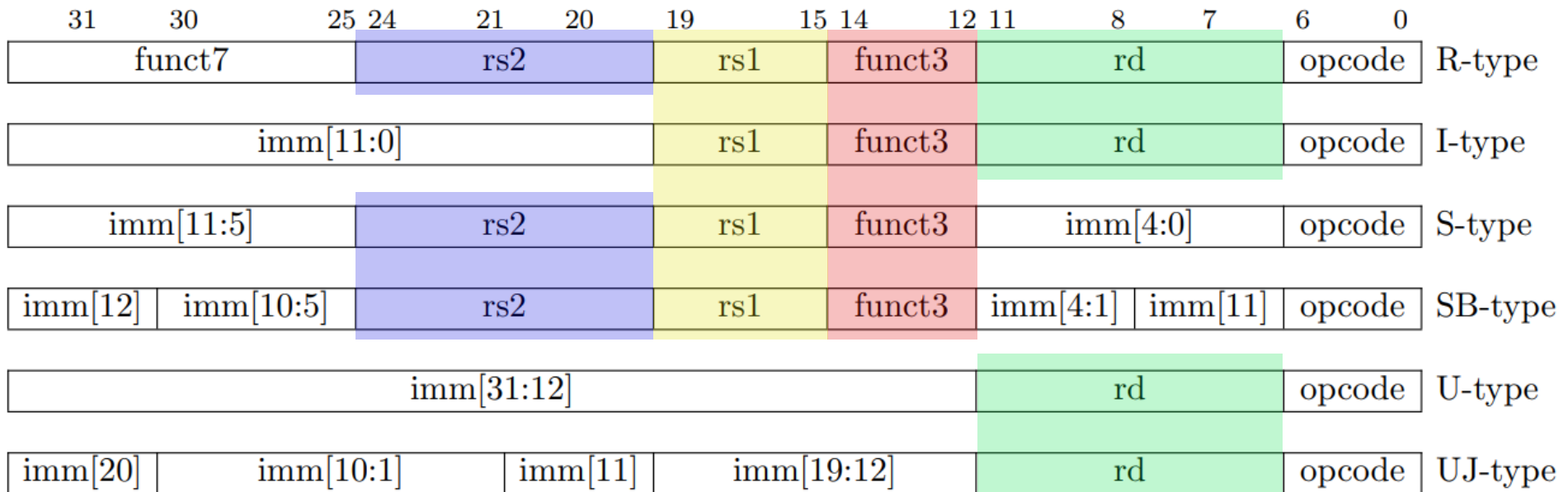


- ❑ Make the common case fast!
 - 12~16 bits capture most cases



Design consideration: Consistent operand encoding location

- Simplifies circuits, resulting in less chip resource usage



CS152: Computer Systems Architecture

Storytime – x86 And Surrounding History

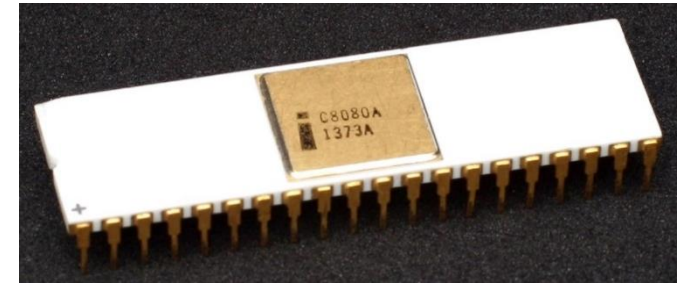


Sang-Woo Jun

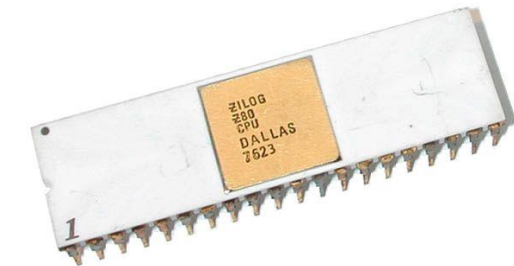
Winter 2022

x86: Evolution with backward compatibility

- ❑ 8080 (1974): 8-bit microprocessor
 - Accumulator, plus 3 index-register pairs
 - Widely successful, spawned many clones
 - Zilog Z80 still manufactured today!
- ❑ Intel iAPX 432 (1975): First 32-bit architecture
 - New ISA, not backwards compatible
 - High-level language features built in
 - Memory access control, garbage collection, etc in hardware
 - No explicit registers, stack-based
 - Bit-aligned variable-length ISA
 - Circuit too large! Spread across two chips
 - ...Slow...



Intel 8080 (Photo Konstantin Lanzet)



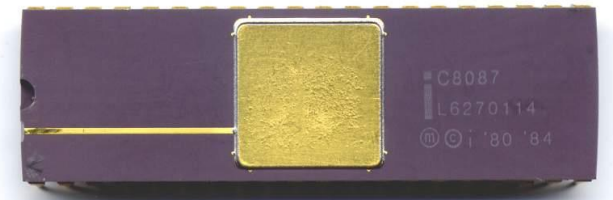
Zilog Z80 (Photo Gennadiy Shvets)



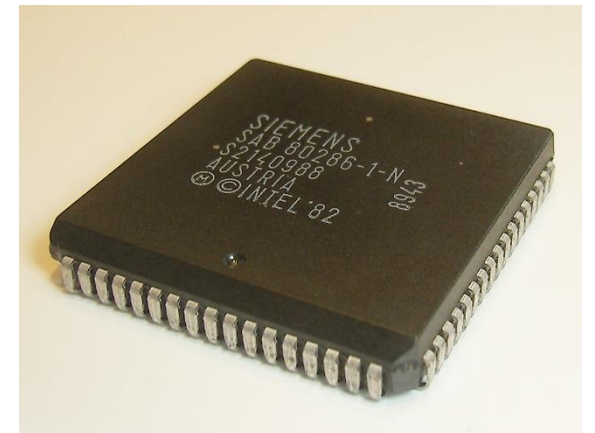
Toshiba Z84C00 (Photo Dharm77, Wikipedia)

x86: Evolution with backward compatibility

- ❑ 8086 (1978): 16-bit extension to 8080
 - Intended temporary substitute until the delayed iAPX 432 became available
 - Backwards compatible with 8080
 - Complex instruction set (CISC)
- ❑ 8087 (1980): floating-point coprocessor
 - Adds FP instructions and register stack
- ❑ 80286 (1982): 24-bit addresses, MMU
 - Segmented memory mapping and protection
 - Each segment was a 16-bit address space
 - Compatibility with legacy programs (CP/M, etc)



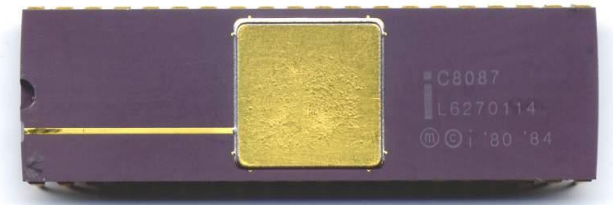
Intel 8087 (Photo Dirk Oppelt)



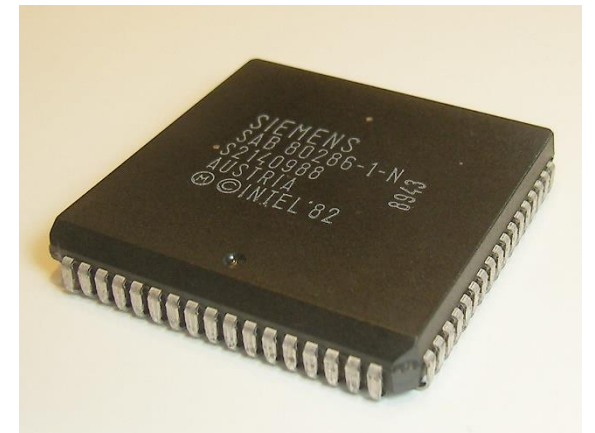
Intel 80286 (Photo Peter Binter)

x86: Evolution with backward compatibility

- ❑ 80386 (1985): 32-bit extension (now IA-32)
 - Additional addressing modes and operations
 - Paged memory mapping as well as segments
 - “Virtual 8086 mode”
 - Special operation mode for a task/process
 - Hardware virtualization support for legacy software (e.g., MS-DOS)
 - Multiple instances of DOS programs could run in parallel
 - OS can finally move beyond MS-DOS!
 - Previously stuck because DOS compatibility could not be ignored
 - DOS software expected exclusive hardware control...
 - Windows 3.1 built on this



Intel 8087 (Photo Dirk Oppelt)



Intel 80286 (Photo Peter Binter)

x86: Evolution with backward compatibility

❑ Further single-thread evolution...

- i486 (1989): pipelined, on-chip caches and FPU
 - Compatible competitors: AMD, Cyrix, ...
- Pentium (1993): superscalar, 64-bit datapath
 - Later versions added MMX (Multi-Media eXtension) instructions
 - The infamous FDIV bug
- Pentium Pro (1995), Pentium II (1997)
 - New microarchitecture (see Colwell, *The Pentium Chronicles*)
- Pentium III (1999)
 - Added SSE (Streaming SIMD Extensions) and associated registers
- Pentium 4 (2001)
 - New microarchitecture
 - Added SSE2 instructions

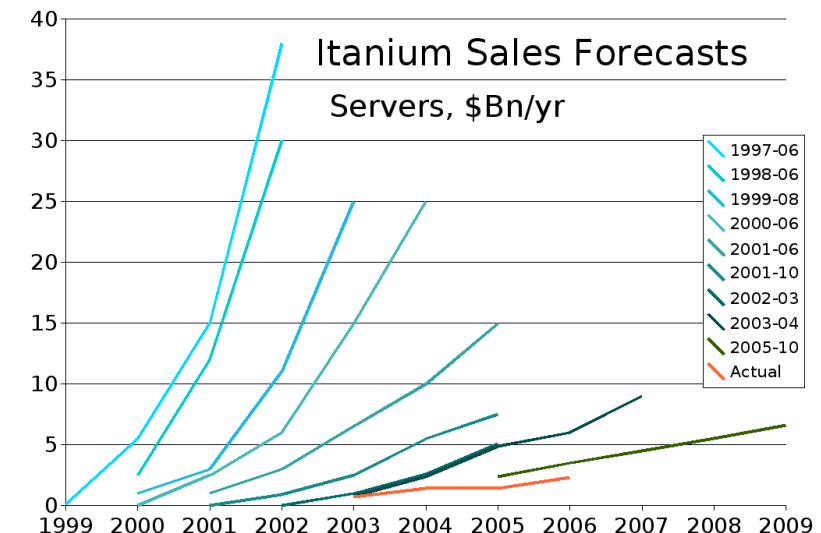


Intel Pentium II (Photo Asimzb, Wikipedia)

x86: Evolution with backward compatibility

❑ Intel Itanium/EPIC (Explicitly Parallel Instruction Computing) – IA-64

- Time to go beyond 32 bits and its 4 GB memory address limitation!
- Time to go beyond 8080 backwards compatibility!
- Time to go beyond CPI of 1!
- “VLIW (Very Long Instruction Word)”
 - Each instruction (“bundle”) consists of three sub-instructions
 - Three instructions issued at once (CPI of 1/3 if lucky)
 - Lots of tricks to deal with data dependencies
 - Difficult design! Delay...
 - Some opinions: Writing compilers was hard...



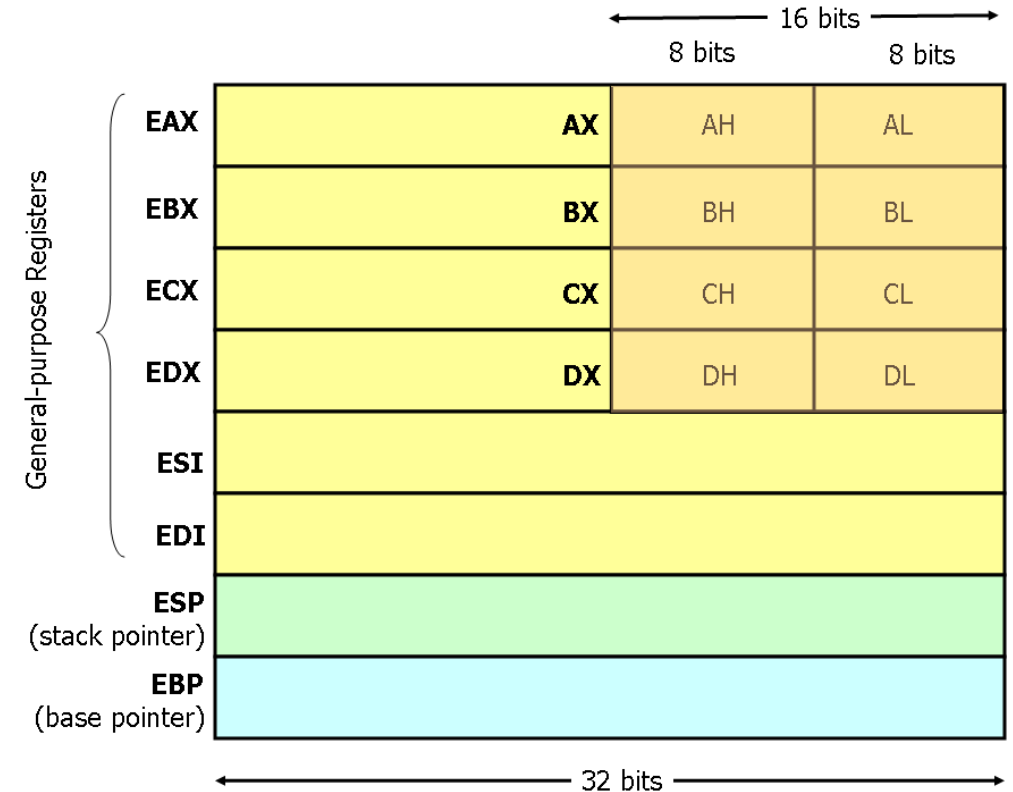
x86: Evolution with backward compatibility

- ❑ Meanwhile at AMD: AMD64, or x86-64
 - Backwards compatible architecture extension to 64 bits
 - Later also adopted by Intel
- ❑ Intel Core (2006) – Going dual-core
 - Added SSE4 instructions, virtual machine support
- ❑ AMD64 (announced 2007): SSE5 instructions
 - Intel declined to follow, instead...
- ❑ Advanced Vector Extension (announced 2008)
 - Longer SSE registers, more instructions

- ❑ If Intel didn't extend with compatibility, its competitors would!
 - Technical elegance ≠ market success

Intel x86 – Registers

- ❑ Much smaller number of registers compared to RISC-V
- ❑ Four ‘general purpose’ registers
 - Naming has historical reasons
 - Originally AX...DX, but ‘Extended’ to 32 bits
 - 64 bit extensions with ‘R’ prefix
- ❑ Aside: Now we know four is too little...
- ❑ Special registers for stack management
 - RISC-V has no special register (Except x0)



Aside: Intel x86 – Addressing modes

- Typical x86 assembly instructions have many addressing mode variants

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- e.g., ‘add’ has two input operands, storing the add in the second

```
add <reg>, <reg>
add <mem>, <reg>
add <reg>, <mem>
add <imm>, <reg>
add <imm>, <mem>
```

Examples

add \$10, %eax — EAX is set to EAX + 10

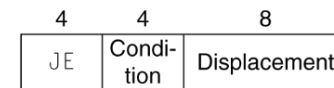
addb \$10, (%eax) — add 10 to the single byte stored at memory address stored in EAX

Aside: Intel x86 – Encoding

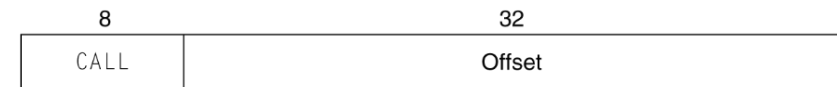
- ❑ Many many complex instructions
 - Fixed-size encoding will waste too much space
 - Variable-length encoding!
 - 1 byte – 15 bytes encoding
- ❑ Complex decoding logic in hardware
 - Hardware translates instructions to simpler micro operations
 - Simple instructions: 1–1
 - Complex instructions: 1–many
 - Microengine similar to RISC
 - Market share makes this economically viable

Comparable performance to RISC!
Compilers avoid complex instructions

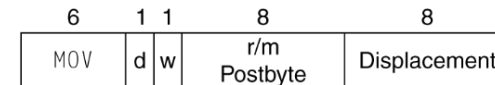
a. JE EIP + displacement



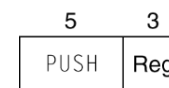
b. CALL



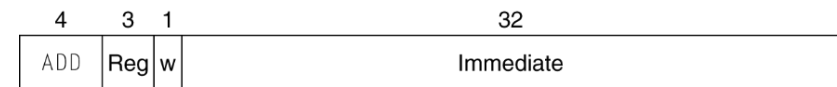
c. MOV EBX, [EDI + 45]



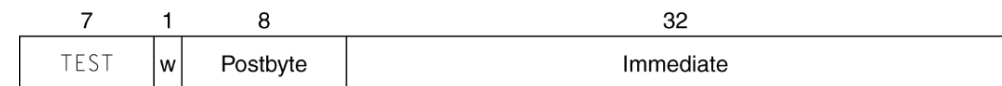
d. PUSH ESI



e. ADD EAX, #6765

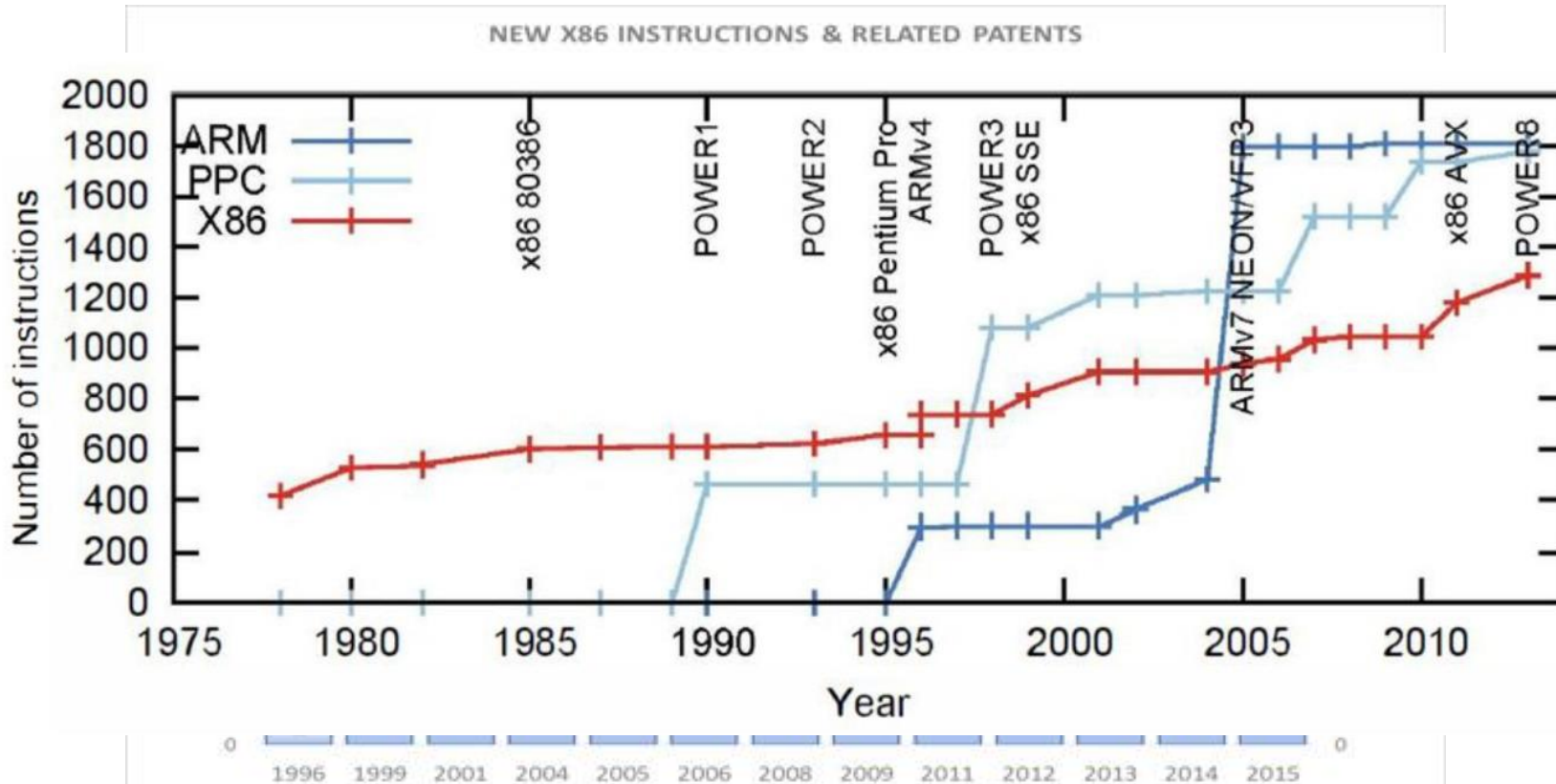


f. TEST EDX, #42



Aside: x86 – Instruction accumulation

- Backward compatibility \Rightarrow instruction set doesn't change
 - But they do accrete more instructions



CS152: Computer Systems Architecture Programming With RISC-V Assembly



Sang-Woo Jun

Winter 2022



Large amount of material adapted from MIT 6.004, “Computation Structures”,
Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”,
and CS 152 Slides by Isaac Scherson

Pseudoinstructions

- ❑ Using raw RISC-V instructions is complicated
 - e.g., How can I load a 32-bit immediate into a register?
- ❑ Solved by “Pseudoinstructions” that are not implemented in hardware
 - Assembler expands it to one or more instructions

Pseudo-Instruction	Description
li dst, imm	Load immediate
la dst, label	Load label address
bgt, ble, bgtu, bleu, ...	Branch conditions translated to hardware-implemented ones
jal label	jal x1, 0(label)
ret	Return from function (jalr x0, x1, 0)


...and more! Look at provided ISA reference

Why x0, why x1?



RISC-V register conventions

- ❑ Convention: Not enforced by hardware, but agreed by programmers
 - Except x0 (zero). Value of x0 is always zero regardless of what you write to it
 - Used to discard operations results. e.g., jalr x0, x1, 0 ignores return address

Registers	Symbolic names	Description	Saver  ?
x0	zero	Hardwired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporary registers	Caller
x8-x9	s0-s1	Saved registers	Callee
x10-x11	a0-a1	Function arguments and return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporary registers	Caller

 Symbolic names also used in assembler syntax

Calling conventions and stack

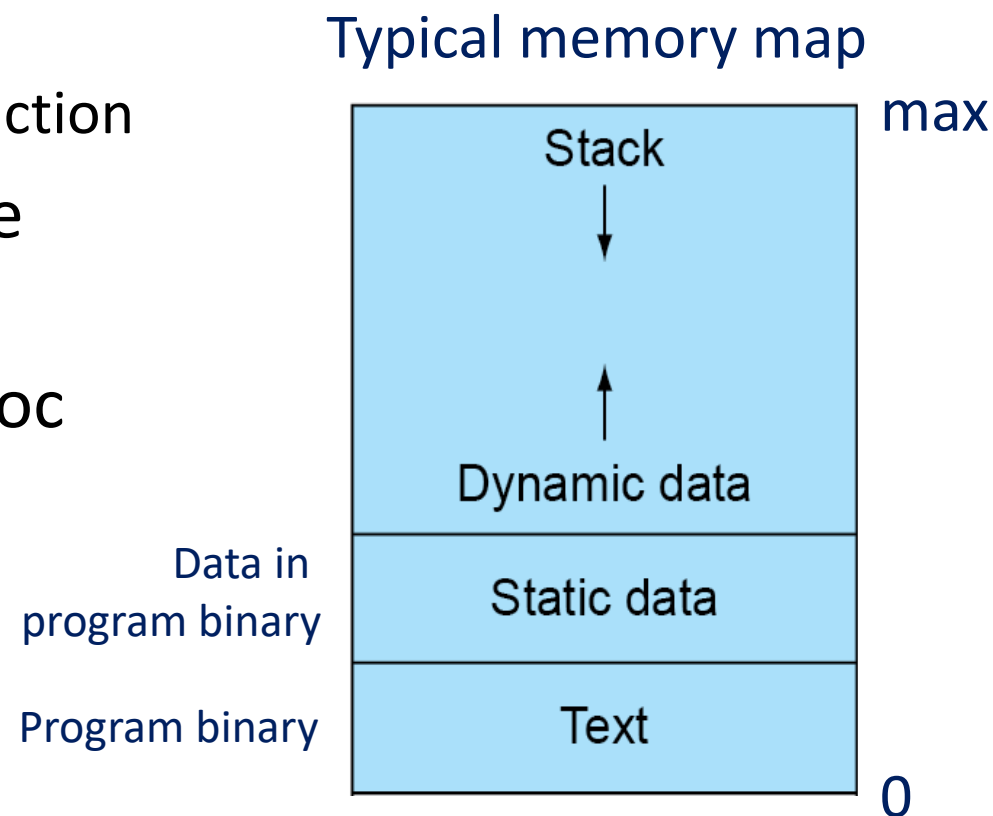
□ Some register conventions during function call

- ra (x1): typically holding return address
 - Saver is “caller”, meaning a function caller must save its ra somewhere before calling
- sp (x2): typically used as stack pointer
- t0-t6: temporary registers
 - Saver is “caller”, meaning a function caller must save its values somewhere before calling, if its values are important (Callee can use it without worrying about losing value)
- a0-a7: arguments to function calls and return value
 - Saver is “caller”
- s0-s11: saved register
 - Saver is “callee”, meaning if a function wants to use one, it must first save it somewhere, and restore it before returning

“Save” where? Registers are limited

Calling conventions and stack

- ❑ Registers saved in off-chip memory across function calls
- ❑ Stack pointer x2 (sp) used to point to top of stack
 - sp is callee-save
 - No need to save if callee won't call another function
- ❑ Stack space is allocated by decreasing value
 - Referencing done in sp-relative way
- ❑ Aside: Dynamic data used by heap for malloc



Example: Using callee-saved registers

❑ Will use s0 and s1 to implement f

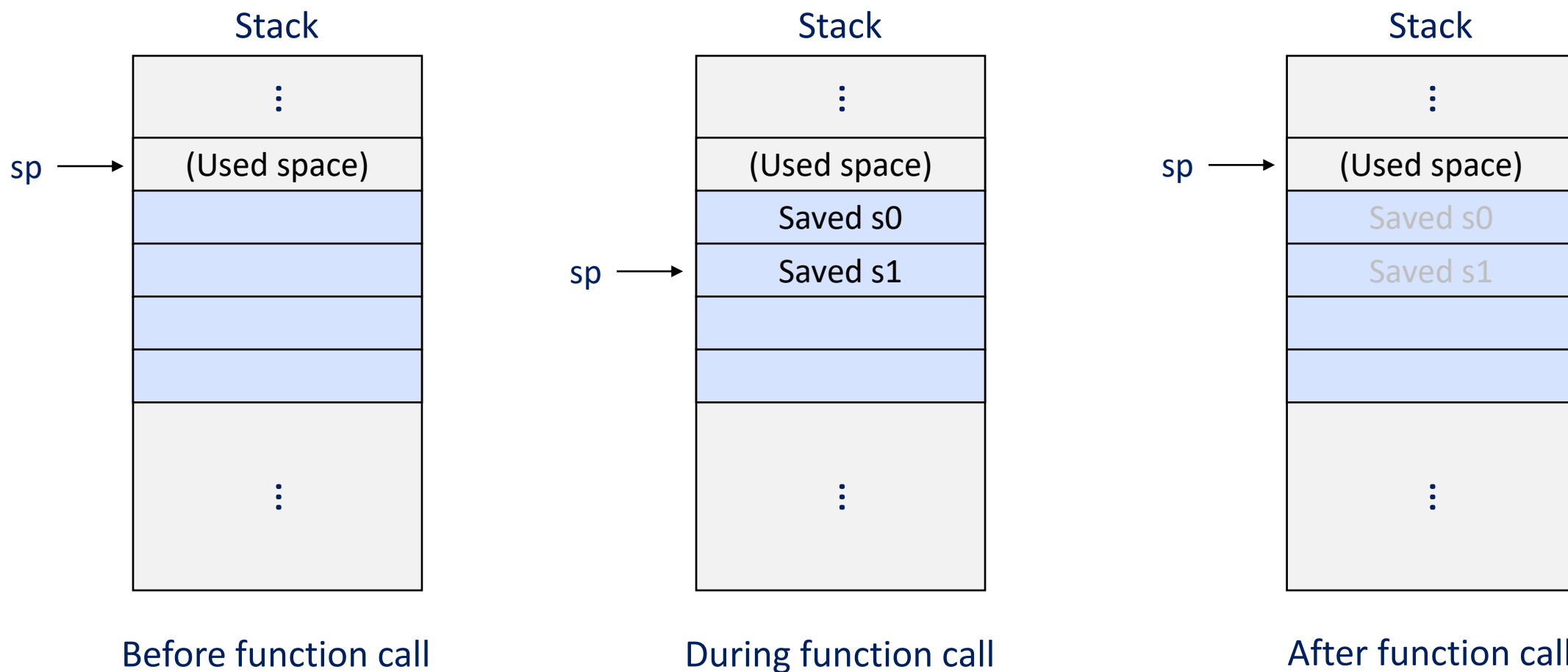
```
int f(int x, int y) {  
    return (x + 3) | (y + 123456);  
}
```

f:

```
addi sp, sp, -8    // allocate 2 words (8 bytes) on stack  
sw s0, 4(sp)      // save s0  
sw s1, 0(sp)      // save s1  
addi s0, a0, 3  
li s1, 123456  
add s1, a1, s1  
or a0, s0, s1  
lw s1, 0(sp)      // restore s1  
lw s0, 4(sp)      // restore s0  
addi sp, sp, 8    // deallocate 2 words from stack  
// (restore sp)
```

ret

Example: Using callee-saved registers



Example: Using caller-saved registers

Caller

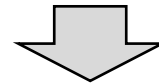
```
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);
```



```
li a0, 1
li a1, 2
addi sp, sp, -8
sw ra, 0(sp)
sw a1, 4(sp) // save y
jal ra, sum
// a0 = sum(x, y)
lw a1, 4(sp) // restore y
jal ra, sum
// a0 = sum(z, y)
lw ra, 0(sp)
addi sp, sp, 8
```

Callee

```
int sum(int a, int b) {
    return a + b;
}
```



```
sum:
    add a0, a0, a1
    ret
```

ra is saved, meaning even if callee calls another function, caller can still retrieve its ra

Why did the caller save a1?
We don't know which registers callee will use
Caller must save all caller-save registers it cares about

Rule of thumb for register conventions

- ❑ Assume function “foo” calls function “bar”
- ❑ There are two sets of general purpose registers, t’s (t0-t6) and s’s (s0-s11)
 - Saved registers (s’s) are callee-save, meaning “bar” must store them somewhere if it wants to use some
 - Temporary registers (t’s) are caller-save, meaning “foo” must save them somewhere if it wants their values to be the same after returning from “bar”
- ❑ Argument registers (a’s) are caller-save
 - If “bar” wants to call another function “bar2”, it must save the a’s it was given, before setting them to its own arguments (which is natural)

Rule of thumb for register conventions

❑ Rule of thumb for saved registers

- For computation ongoing across function (“bar”) calls, use s’s
- **Simple to just use s’s for most register usage**
- Each function (“bar”) stores all s’s (it plans to use) in the stack at beginning, and restore them before returning

❑ Rule of thumb for temporary registers

- Use t’s for intermediate values that are no longer important after the function call, for example calculating arguments for “bar”.
- “Foo” must store t’s in stack (if it wants their values to persist) before calling “bar”, but **simpler to just restrict use of t’s for values we don’t expect to persist**

In reality, compilers do this for us `~_(\ツ)_/~`

Aside: Handling I/O

- ❑ How can a processor communicate with the outside world?
- ❑ Special instructions? Sometimes!
 - RISC-V defines CSR (Control and Status Registers) instructions
 - Check processor capability (I/M/E/A/..?), performance counters, system calls, ...
 - “Port-mapped I/O”
- ❑ E.g., x86 has “IN”, “OUT” instructions
 - Goes back to how 8080 did I/O
 - “IN \$0x60, %al” reads a keyboard input from the PS/2 controller



Source: Wikipedia

Aside: Handling I/O

- ❑ For efficient communication, memory-mapped I/O
 - Happens outside the processor
 - I/O device directed to monitor CPU address bus, intercepting I/O requests
 - Each device assigned one or more memory regions to monitor
 - Some memory commands handles by memory, some by peripherals!

Example:

In the original Nintendo GameBoy, reading from address 0xFF00 returned a bit mask of currently pressed buttons

Both approaches require one CPU instruction per word I/O...

Aside: Handling I/O

❑ Even faster option: DMA (Direct Memory Access)

- Off-chip DMA Controller can be directed to read/write data from memory without CPU intervention
- Once DMA transfer is initiated, CPU can continue doing other work
- Used by high-performance peripherals like PCIe-attached GPUs, NICs, and SSDs
 - Hopefully we will have time to talk about PCIe!
- Contrast: Memory-mapped I/O requires one CPU instruction for one word of I/O
 - CPU busy, blocking I/O hurts performance for long latency I/O

Wrapping up...

❑ Design principles

1. Simplicity favors regularity
2. Smaller is faster
3. Good design demands good compromises

❑ Make the common case fast

❑ Powerful instruction \nRightarrow higher performance

- Fewer instructions required, but complex instructions are hard to implement
 - May slow down all instructions, including simple ones
- Compilers are good at making fast code from simple instructions