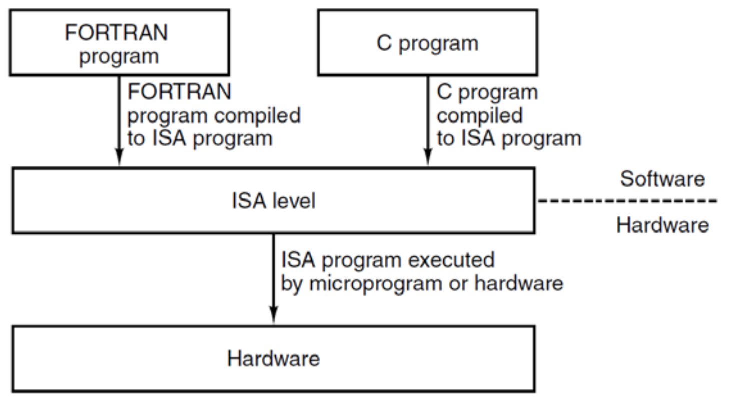
Instruction Set Architecture (ISA) Level

- Interface between software and hardware


FORTRAN program
C program
FORTRAN program compiled to ISA program
C program compiled to ISA program
ISA level
Software
Hardware
ISA program executed by microprogram or hardware
Hardware

Properties of ISA level

- ISA level code is what a compiler outputs
- To produce ISA code, compiler writer has to know
  - the memory model
  - registers available
  - data types and instructions available

ISA level registers

- Limited number of registers
- Main function of ISA level registers:
  - provide rapid access to heavily used data
- Examples (core-i7 registers)
  - EAX: main arithmetic register
  - EIP: program counter

Data Types

- Numeric data types
  - Unsigned integers
  - Signed integers (modern PCs use two's complement form)
  - Floating-point (modern PCs use IEEE 754 standard)
- Nonnumeric data types
  - Common character codes: ASCII and UNICODE
  - Boolean type

Instructions

- Main feature of ISA level is its set of machine instructions
- They control what the machine can do

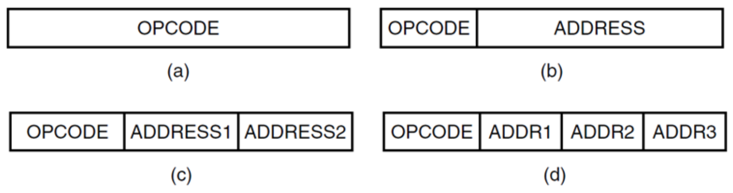
Example (core-i7 instructions):

- ADD (addition)
  - ADD DST, SRC // DST $\leftarrow$ DST + SRC
- SUB (subtraction)
  - SUB DST, SRC // DST $\leftarrow$ DST - SRC

Instruction Formats

- An instruction consists of an **opcode**, plus additional information such as where operands come from, where results go to
- Opcode tells what instruction does

Some common instruction formats



Addressing

- Subject of specifying where the operands (addresses) are
  - Core-i7 ADD instruction requires 2 operands, and instruction must tell where to find operands and where to put result

Addressing Modes

- Methods of interpreting the bits of an address field to find operand

Immediate Addressing

- Simplest way to specify where the operand is
- Address part of instruction contains operand itself (**immediate operand**)
- Operand is automatically fetched from memory at the same time the instruction itself is fetched
  - immediately available for use
- No additional memory references are required
- Disadvantages
  - only a constant can be supplied
  - value of the constant is limited by the size of address field
- Good for specifying small integers

Example (core-i7):
MOV R1, 8 // R1 ← 8

Direct Addressing

- Operand is in memory, and is specified by giving its full address (memory address is hardwired into instruction)
- Instruction will always access exactly same memory location

- Can only be used for global variables whose address is known at compile time

Example (core-i7):
ADD R1, (100) // R1 ← R1 + Mem[100]

Register Addressing

- Same as direct addressing with the exception that it specifies a register instead of memory location
- Most common addressing mode on most computers since register accesses are very fast
- Compilers try to put most commonly accessed variables in registers

Example (core-i7):
SUB R1, R2 // R1 ← R1 – R2

Register Indirect Addressing

- Operand being specified comes from memory or goes to memory
- Its address is not hardwired into instruction, but is contained in a register Can reference memory without having full memory address in the instruction

- Different memory words can be used on different executions of the instruction

Example (core-i7):
ADD R1, (R2) // R1 ← R1 + Mem[R2]

Indexed Addressing

- Memory is addressed by giving a register plus a constant offset
- Used to access local variables

Example:
ADD R1, 100(R2) // R1 ← R1 + Mem[100+R2]

Example ISAs

- MIPS
  - Nowadays limited to embedded devices
- RISC-V
  - An open instruction set (adheres to RISC principle)
  - Introduced in 2010
  - Offers 32-bit, 64-bit instructions
  - Open source simulators and compilers are available

Example ISAs

- Intel x86
  - Started more than 40 years ago
  - Used in Intel and AMD processors
- ARM (Acorn RISC Machine)
  - Started more than 35 years ago
  - Popular in battery powered devices
  - Used in desktop processors as well

RISC-V Assembly Language

- 64-bit registers
- Register count = 32
  - `x0 – x31`
  - `x0` contains the value of zero
  - Data must be in registers to perform arithmetic
- Uses byte addressing
- 2^{61} memory words
  - Memory[0], …, Memory[18,446,744,073,709,551,607]
  - Word [32 bits], Double word [64 bits]
- Memory words are accessed only by data transfer instructions

- Data structures (arrays, structures) are kept in memory
- Memory is a large single dimensional array
- Each line can contain at most one instruction
- Text to the right of // are comments
- Comments always terminate at the end of the line

Arithmetic instructions

- All arithmetic instructions have 3 parameters
  - *add* (addition)
  - *sub* (subtraction)
  - *addi* (add immediate)(used to add constants)

Example:

```
• add x21, x22, x23          // x21 ← x22 + x23
• sub x21, x22, x23          // x21 ← x22 – x23
• addi x21, x22, 20          // x21 ← x22 + 20
```

Example:

Translate the following C assignment statement to RISC-V assembly language

```
    f = (a + b) + 15;
```

Answer:

Suppose the variables *a*, *b* are assigned to registers `x21, x22` respectively

```
  add x23, x21, x22
  addi x24, x23, 15
```

Example:

What is RISC-V compiled code for the following C assignment statement?

```
    f = (a + b) - (c + d);
```

Answer:

Suppose `a, b, c, d` are assigned to `x21, x22, x23, x24`.

```
add x25, x21, x22
add x26, x23, x24
sub x25, x25, x26
```

Data transfer instructions

- Transfer data between memory and registers
  - *lw* (loads a word from memory to register)
  - *sw* (stores a word from register to memory)
  - *ld* (loads a double word from memory to register)
  - *sd* (stores a double word from register to memory)

Example:

```
• ld x21, 40(x22)          // x21 ← Memory[x22 + 40]
• sw x21, 40(x22)          // Memory[x22 + 40] ← x21
```

Example:

Let *A* be an array of 100, 8-byte integers. Base address of the array is `x20`. Compile the following C assignment statement to RISC-V.

```
    f = b + A[6];
```

Answer:

Suppose *b* is assigned to `x21`.

Note: arrays are stored in consecutive memory locations

```
ld x29, 48(x20)
add x29, x21, x29
```

Example:

Let *A* be an array of 100, 8-byte integers with the base address `x20`. What is the RISC-V compiled code for the following C assignment statement?

```
    A[10] = A[6] + 5;
```

Answer:

ld x29, 48(x20)
addi x29, x29, 5
sd x29, 80(x20)

RISC-V Machine Language

- All RISC-V instructions are 32 bits long
- Register mapping:
  - `x0 – x31` (registers 0 – 31)
- Instruction formats
  - R-type (for register operations)
  - I-type (for add immediate and load instructions)
  - S-type (for store instructions)

RISC-V instruction format (R-type)

*funct*7	*rs*2	*rs*1	*funct*3	*rd*	*opcode*
7-bit	5-bit	5-bit	3-bit	5-bit	7-bit

- *opcode*: tells what the instruction does
- *rd*: register destination operand
- *funct*3: additional opcode field

- *rs*1: first register source operand
- *rs*2: second register source operand
- *funct*7: additional opcode field

RISC-V instruction encoding

Instruction	Type	*funct7*	*rs2*	*rs*1	*funct3*	*rd*	*opcode*
add	R	0	reg	reg	0	reg	51
sub	R	32	reg	reg	0	reg	51

- reg: register number (0-31)

Example:
Write the RISC-V machine language code of the following C assignment statement *f* = *a* + *b*;

Answer:
Suppose *a* and *b* are assigned to x21 and x22, and *f* to be stored in x23.
RISC-V assembly code: *add* x23, x21, x22

funct7	*rs2*	*rs*1	*funct3*	*rd*	*opcode*
0	22	21	0	23	51

Machine language code:
0000000 10110 10101 000 10111 0110011

RISC-V instruction format (I-type)

immediate	*rs*1	*funct*3	*rd*	*opcode*
12-bit	5-bit	3-bit	5-bit	7-bit

- *immediate*: two's complement value
  - Byte offset (address) for load instructions
  - Constant for add immediate instructions

RISC-V instruction encoding

Instruction	Type	*immediate*	*rs*1	*funct*3	*rd*	*opcode*
addi	I	constant	reg	0	reg	19
ld	I	address	reg	3	reg	3

Example:
Let *A* be an array of 100, 8-byte integers with the base address x20. Convert C assignment statement *f* = *A*[6] + 5; to RISC-V machine language.

Answer:

Suppose *f* to be stored in x29.

```
ld x29, 48(x20)
addi x29, x29, 5
```

instruction	*immediate*	*rs1*	*funct3*	*rd*	*opcode*
ld	48	20	3	29	3
addi	5	29	0	29	19

Machine language code:

000000110000 10100 011 11101 0000011
000000000101 11101 000 11101 0010011

RISC-V instruction format (S-type)

immediate(11:5)	*rs2*	*rs1*	*funct3*	*immediate*(4:0)	*opcode*
7-bit	5-bit	5-bit	3-bit	5-bit	7-bit

- *Immediate* (12 bits): represents address offset
  - Splits into 2 fields (lower 5 bits, upper 7 bits)
  - Upper bits represents the quotient of offset when divided by 2^5
  - Lower bits represents the remainder of offset when divided by 2^5

RISC-V instruction encoding

Instruction	Type	*immediate*	*rs2*	*rs1*	*funct3*	*immediate*	*opcode*
sd	S	address	reg	reg	3	address	35

Example:

Let *A* be an array of 100, 8-byte integers with the base address x20. What is the RISC-V machine language code of the following C assignment statement?

```
A[10] = b + A[6];
```

b is assigned to x21

Answer:

```
ld x29, 48(x20)
add x29, x21, x29
sd x29, 80(x20)
```

instruction	*immediate*		*rs1*	*funct3*	*rd*	*opcode*
ld	48		20	3	29	3
Instruction	*funct7*	*rs2*	*rs1*	*funct3*	*rd*	*opcode*
add	0	29	21	0	29	51

Instruction	*immediate*	*rs2*	*rs1*	*funct3*	*immediate*	*opcode*
sd	2	29	20	3	16	35

Machine language code:

000000110000 10100 011 11101 0000011
0000000 11101 10101 000 11101 0110011
0000010 11101 01100 011 10000 0100011

RISC-V Shift Instructions

- Shift left logical immediate
  `slli x21, x22, 4`
  (all bits in the double word x22 are shifted 4 positions to the left and the result is stored in x21)
- Shift right logical immediate
  `srli x21, x22, 3`
  (all bits in x22 are shifted 3 positions to the right and the result is stored in x21)

Example:

```
slli x21, x22, 4
     x22 = 0000……0000 0000 1100       // decimal 12

     x21 = 0000……0000 1100 0000       // decimal 192
```

RISC-V Logical Instructions

- AND (bit-by-bit)
  - `and x21, x22, x23      // x21 ← (x22 & x23)`
- Inclusive OR (bit-by-bit)
  - `or x21, x22, x23       // x21 ← (x22 | x23)`
- Exclusive OR (bit-by-bit)
  - `xor x21, x22, x23      // x21 ← (x22 ^ x23)`

Example:

```
and x21, x22, x23
     x22 = 0000……0000 0010 1110 1100
     x23 = 0000……0000 0100 1011 0110

     x21 = 0000……0000 0000 1010 0100
```

Example:

```
xor x21, x22, x23
     x22 = 0000……0000 0010 1110 1100
     x23 = 0000……0000 0100 1011 0110

     x21 = 0000……0000 0110 0101 1010
```

RISC-V Conditional Branch Instructions

- Branch if equal
  - `beq x21, x22, Label1`
    `// if (x21 == x22) go to Label1`
- Branch if not equal
  - `bne x21, x22, Label2`
    `// if (x21 ≠ x22) go to Label2`

Example:

Compile the following C if-else statement to RISC-V.

```
     if (i == j) f = g + h; else f = g – h;
```

`f,g,h,i,j are assigned to x21, x22, x23, x24, x25`

```
bne x24, x25, Else          // go to Else if i ≠ j)
add x21, x22, x23           // f = g + h
beq x0, x0, Exit            //go to Exit
Else: sub x21, x22, x23     // f = g - h
Exit:
```

Example:

Let A be an array of 100, 8-byte integers with the base address `x20`. Convert the following C while loop to RISC-V assembly language.

```
     while (A[i] == n)
          i += 1;
i,n are assigned to x21, x22
```

```
Loop: slli x19, x21, 3      // x19 = i*8
add x19, x19, x20           // x19 = address of A[i]
ld x29, 0(x19)              // x29 = A[i]
bne x29, x22, Exit          // go to Exit if A[i]≠n
addi x21, x21, 1            // i = i+1
beq x0, x0, Loop            // go to Loop
Exit:
```