Lecture 2: Instruction Semantics

Spring 2024
Jason Tang

Topics

- CPU operations
- Instruction types
- Functional calls

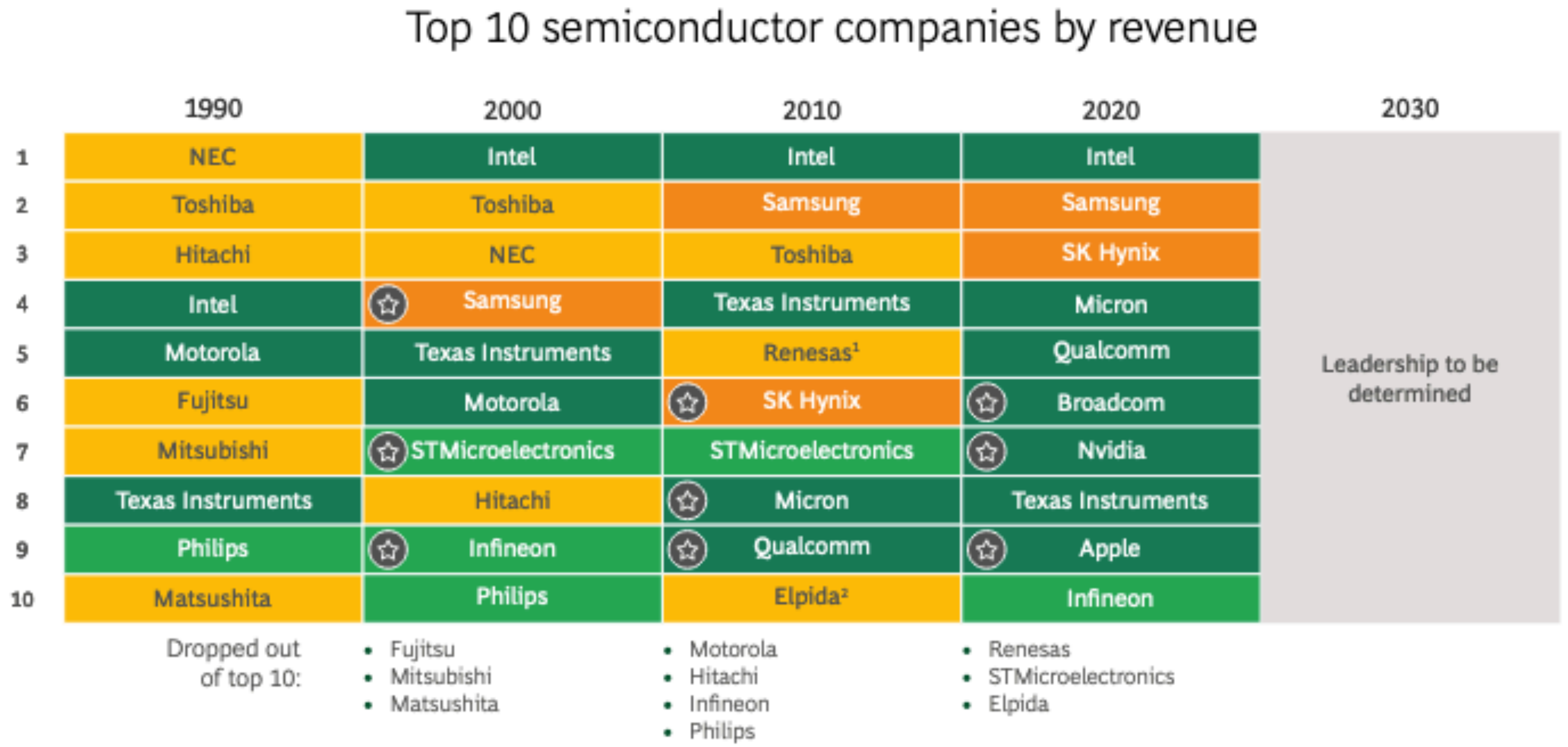
Instructions

- Although there are many different CPU designs, they all share similar instruction sets:
    - Fundamental operations that all computers must provide
    - Designers have same goal of finding a machine language that maximizes performance while minimizing cost
- This class is based around the ARMv8-A design
    - Designed by Arm holdings, a British semiconductor company
    - Used in nearly all modern smartphones, tablets, and smartwatches

Instruction Set Architecture (ISA)

- ISA: abstract interface between hardware and lowest level of software, encompassing all necessary information to write and run a machine language program
  - Includes instructions, registers, memory access, I/O, etc.
  - Enables different *implementations* of ISA to run the same software
- Application Binary Interface (ABI): user portion of instruction set that defines how software should use the ISA to pass data between themselves and the underlying operating system
  - Defines a standard for binary portability across computers, OS revisions, and compiler revisions

Changes in CPU Popularity



Top 10 semiconductor companies by revenue

	1990	2000	2010	2020	2030
1	NEC	Intel	Intel	Intel	
2	Toshiba	Toshiba	Samsung	Samsung	
3	Hitachi	NEC	Toshiba	SK Hynix	
4	Intel	☆ Samsung	Texas Instruments	Micron	
5	Motorola	Texas Instruments	Renesas[1]	Qualcomm	Leadership to be determined
6	Fujitsu	Motorola	☆ SK Hynix	☆ Broadcom	
7	Mitsubishi	☆ STMicroelectronics	STMicroelectronics	☆ Nvidia	
8	Texas Instruments	Hitachi	☆ Micron	Texas Instruments	
9	Philips	☆ Infineon	☆ Qualcomm	☆ Apple	
10	Matsushita	Philips	Elpida[2]	Infineon	
	Dropped out of top 10:	• Fujitsu • Mitsubishi • Matsushita	• Motorola • Hitachi • Infineon • Philips	• Renesas • STMicroelectronics • Elpida	

- In the past ten years, the shift has been from high performance computers to handheld and embedded systems

https://www.semiconductors.org/wp-content/uploads/2022/11/2022_The-Growing-Challenge-of-Semiconductor-Design-Leadership_FINAL.pdf

Operations of Computer Hardware

- Assembly language is a symbolic representation of what the processor actually understands
- Different assembly language syntaxes can exist for the same ISA
  - See AT&T vs Intel debate
- In ARMv8-A, each assembly line translates to a single instruction
- Example C code and resulting assembly:

```
f = (g + h) - (i + j)
```

```
add     w1, w1, w0
add     w0, w2, w3
sub     w0, w1, w0
```

Operands of Computer Hardware

- Registers are the bricks of computer construction
  - Hardware design primitives visible to programmers
- Size and number of registers differ by architectures
  - For ARMv8-A, the 31 general purpose registers are 64 bits wide
  - When referring to the lower 32 bits, use notation **w*n***; to refer to all 64 bits use **x*n***
- For many operations, the destination register is listed first

```
add     w1, w1, w0
add     w0, w2, w3
sub     w0, w1, w0
```

Common Addressing Modes

- Direct (or immediate): value encoded in instruction
- Register: value stored within register
- Register indirect: value stored in memory at the address given by a register
- Register displacement: base (a register) + offset (an immediate value)
    - Scaled displacement: base + offset × scale
- PC-relative: Program Counter + offset × scale

Complex Data Representation

- Given limited number of registers, remaining data (e.g., structs and arrays) are kept in memory
- Data transfer instructions to move data in memory to / from a register are traditionally called load / store
- Accessing a specific memory address is done indirectly, via register displacement
- Example in C: `g = h + A[7]`
  - Let `A` be an array of `int`s (32-bits each), and its starting address is in register w0
  - Then resulting assembly is:

```
ldr     w0, [x0, #28]
add     x2, x0, x1
```

Compilation and Address Formation

- Compiler keeps frequently used data values in registers
  - In many architectures (including ARMv8-A), arithmetic operands *must* be a register, not a location in memory
  - Thus compiler keeps frequently accessed values in registers
  - Stores everything else (including larger data structures and arrays) in main memory
- Address in memory = starting address + (index × element size)

```
int A[] = { … };
… A[7];
```

$\Rightarrow$

```
ldr     w0, [w0, #28]
```

Byte Addresses

- When operating on data that is not word aligned within memory (e.g., an individual byte), need to specify which byte within the word to use
  - Big-endian (also known as network order): Motorola 68k, SuperH
  - Little-endian: x86-64
  - Bi-endian: can switch endianness as necessary
    - PowerPC starts in big-endian, can switch to little-endian
    - ARMv7 and ARMv8-A start in little-endian, can switch to big-endian

Endian Example

- Example: to store the hexadecimal value 0x12345678 in memory starting at address 0x1000:

	memory cells			
	0x1000	0x1001	0x1002	0x1003
big-endian	0x12	0x34	0x56	0x78
little-endian	0x78	0x56	0x34	0x12

- Watch out for "mixed-endian" machines that may have other encoding schemes

- In ARMv8-A, **`ldr`** loads an entire word, while **`ldrb`** loads a single byte

Stored Program Concept

- Modern computers are built on two key principles:
    - Instructions and data are represented as numbers (specifically, binary representation)
    - Instructions and data are stored in memory as numbers
- Compiler transforms code into instructions (code generation), and then the assembler translates instructions (assembles) into underlying numeric representation for that instruction set architecture

```
f = (g + h) - (i + j)
```

```
add     w1, w1, w0
add     w0, w2, w3
sub     w0, w1, w0
```

```
add     w1, w1, w0
add     w0, w2, w3
sub     w0, w1, w0
```

```
0x0b000021
0x0b030040
0x0b000020
```

ARMv8-A Instruction Encoding

- In ARMv8-A, [most] instructions are encoded with exactly 32 bits (a fixed instruction length)
    - x86-64 has variable instruction length
- Instruction encoding defines how instructions are numerically represented
    - Example: so-called "register-type" instructions are encoded as:

R-Type	opcode	R_m	shamt	R_n	R_d
	11 bits	5 bits	6 bits	5 bits	5 bits

add w1, w1, w0
(0x0b000021)

00001011000	00000	000000	00001	00001
add	second operand = R0	shift amount = 0	first operand = R1	dest register = R1

ARMv8-A Instruction Encoding

- Data transfers (loads and stores) use similar encoding
  - Offset must be from –256 to +255 (a range of 2^9 bits) of base register

D-Type	opcode	address	op2	R_n	R_t
	11 bits	9 bits	2 bits	5 bits	5 bits

- Immediate instructions contain a direct value instead of a register number
  - Immediate value limited to 12 bits

I-Type	opcode	immediate	R_n	R_d
	10 bits	12 bits	5 bits	5 bits

Unconditional Branching

- Several ways to unconditionally change program flow, like C's **`goto`** statement, or invoking a function
- Use **`b`** to unconditionally jump to a constant address (usually a label)
- Use **`blr`** to invoke a function pointer

```c
void foo(void) {
  /* ... */
}

int main(void) {
  void (*func_ptr)(void) = foo;
  func_ptr();
  return 0;
}
```

```
foo:
    ...
    ret

main:
    ...
    adrp x8, foo
    blr x8
```

Conditional Branching

- Programs also need to make decisions, like C's `if` and `for` statements
- Use `cbz` (compare and branch if zero) and `cbnz` (compare and branch if not zero) to conditionally jump to a label
- Other instructions are `b.gt` (branch if greater than or equal), `b.ne` (branch if not equal)

```
if (x == 0)
  x = 10
else
  x += 20
```

→

```
        cbnz    x0, else
        mov     x0, #10
        b       end_if
else:
        add     x0, x0, #20
end_if:
        /* end of if stmt */
```

Condition Codes

- Many instructions can affect condition codes:

condition bit	set when most recent instruction resulted in:
zero (Z)	equal
carry (C)	carry out of most significant bit
negative (N)	most significant bit was 1
overflow (V)	an overflow

- Use a combination of these codes to express full range of comparisons: less than (`lt`), greater than (`gt`), less than or equal (`le`), greater than or equal (`ge`), equal (`eq`), and not equal (`ne`)
  - Example: `b.lt` to branch if most recent condition resulted in less than

Setting Condition Codes

- Many ARMv8-A arithmetic instructions can have a `s` suffix to set condition code
- Can also directly compare a register to an immediate value via `cmp` instruction
- Example: calculate 3^x using a loop

```
result = 1;
while (x > 0) {
    result *= 3;
    x--;
}
```

```
        mov     x0, #1
        mov     x2, #3
        cbz     x1, end
start:
        mul     x0, x0, x2
        subs    x1, x1, #1
        b.gt    start
end:
        /* end of while */
```

Function Calls

- Calling a function depends upon the calling convention for the language, compiler, and instruction set
  - Need to agree on where store function arguments (register/stack/memory), where to store resulting value, and which registers are to be preserved by the caller and which by the callee
- When updating the calling stack, need to decide who updates the stack pointer (caller or callee) and by how much (prologue and epilogue)
  - For many architectures, the stack must be aligned to some boundary (every 4 bytes or every 8 bytes)

ARMv8-A Procedure Call Standard

- First eight arguments to function are stored in registers **`x0`** through **`x7`**
    - Use stack if more space is needed for incoming parameters
- **`x0`** holds the result of the function
- **`x19`** through **`x29`** are callee-saved
- **`x30`** is the link register
- **`SP`** is the stack register (special register 31)
- All other registers are caller-saved

Procedure Call Standard for the ARM 64-bit Architecture, §5.1.1

Function Prologue and Epilogue

- Callee is responsible to moving stack pointer to hold local variables (prologue) and then cleaning up after itself (epilogue)

- Prologue and epilogue are added by the compiler; they are not hardware enforced

```
static int __attribute__((noinline))
 x(int g, int h, int i, int j) {
    int f = (g + h) - (i + j);
    return f;
}
```

```
x:
  sub    sp, sp, #0x20
  str    w0, [sp, #12]
  str    w1, [sp, #8]
  str    w2, [sp, #4]
  str    w3, [sp]
  ldr    w1, [sp, #12]
  ldr    w0, [sp, #8]
  add    w1, w1, w0
  ldr    w2, [sp, #4]
  ldr    w0, [sp]
  add    w0, w2, w0
  sub    w0, w1, w0
  str    w0, [sp, #28]
  ldr    w0, [sp, #28]
  add    sp, sp, #0x20
  ret
```

Prologue

Epilogue

Calling Functions

- Architectures vary on how to call and return from a function, specifically on where to store/load the return address
  - On x86-64, return address is pushed onto the stack upon `call`. The `ret` instruction pops top-most value from stack and jumps to that address
  - On ARMv8-A, return address is stored in `x30` as a side-effect of `bl`. The `ret` instruction jumps to the address stored in `x30`
- Callee is responsible for saving return address if it is not a leaf function

```
some_func(10, 20, 30, 40);
```

→

```
mov      x0, #0x0a
mov      x1, #0x14
mov      x2, #0x1e
mov      x3, #0x28
bl       some_func
```