15-740/18-740 Computer Architecture Lecture 3: Performance

Prof. Onur Mutlu
Carnegie Mellon University

Last Time …

- Some microarchitecture ideas
  - Part of microarchitecture vs. ISA
- Some ISA level tradeoffs
  - Semantic gap
  - Simple vs. complex instructions -- RISC vs. CISC
  - Instruction length
  - Uniform decode
  - Number of registers

Review: ISA-level Tradeoffs: Number of Registers

- Affects:
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - (uarch) Size, access time, power consumption of register file

- Large number of registers:

  + Enables better register allocation (and optimizations) by compiler → fewer saves/restores

  -- Larger instruction size

  -- Larger register file size

  -- (Superscalar processors) More complex dependency check logic

ISA-level Tradeoffs: Addressing Modes

- Addressing mode specifies how to obtain an operand of an instruction
  - Register
  - Immediate
  - Memory (displacement, register indirect, indexed, absolute, memory indirect, autoincrement, autodecrement, ...)

- More modes:

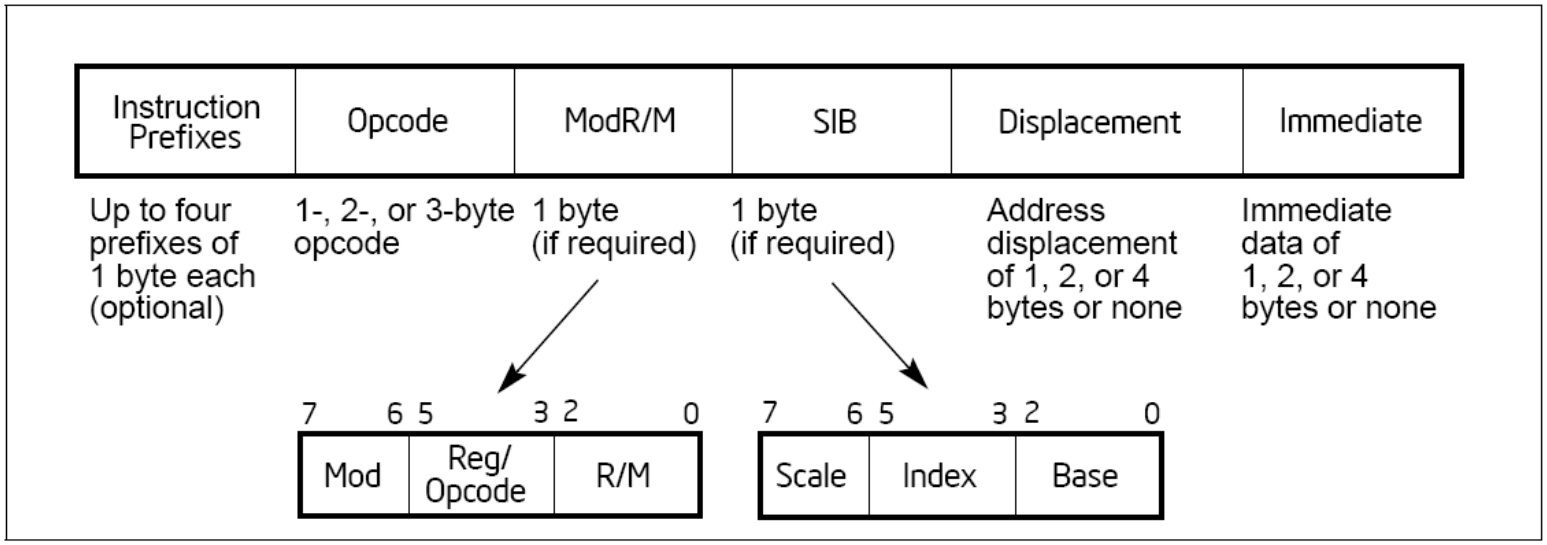
  + help better support programming constructs (arrays, pointer-based accesses)

  -- make it harder for the architect to design

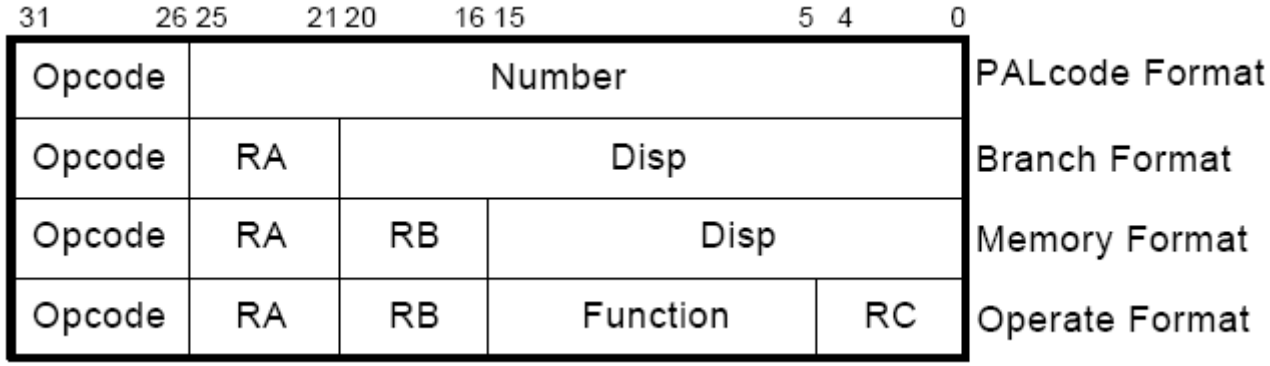
  -- too many choices for the compiler?
    - Many ways to do the same thing complicates compiler design
    - Read *Wulf, "Compilers and Computer Architecture"*

x86 vs. Alpha Instruction Formats

- x86:



- Alpha:



register indirect

absolute

register + displacement

register

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table.

indexed (base + index)

scaled (base + index*4)

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base – (In binary) Base –			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	89	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

NOTES:

1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

MOD bits	Effective Address
00	[scaled index] + disp32
01	[scaled index] + disp8 + [EBP]
10	[scaled index] + disp32 + [EBP]

Other ISA-level Tradeoffs

- Load/store vs. Memory/Memory
- Condition codes vs. condition registers vs. compare&test
- Hardware interlocks vs. software-guaranteed interlocking
- VLIW vs. single instruction
- 0, 1, 2, 3 address machines
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Aligned vs. unaligned access
- Supported data types
- Software vs. hardware managed page fault handling
- Granularity of atomicity
- Cache coherence (hardware vs. software)
- ...

Programmer vs. (Micro)architect

- Many ISA features designed to aid programmers
- But, complicate the hardware designer's job

- Virtual memory
  - vs. overlay programming
  - Should the programmer be concerned about the size of code blocks?
- Unaligned memory access
  - Compile/programmer needs to align data
- Transactional memory?

Transactional Memory

THREAD 1

```
enqueue (Q, v) {
  Node_t node = malloc(…);
  node->val = v;
  node->next = NULL;
  acquire(lock);
  if (Q->tail)
    Q->tail->next = node;
  else
    Q->head = node;
  Q->tail = node;
  release(lock);
}
```

```
begin-transaction
…
enqueue (Q, v); //no locks
…
end-transaction
```

THREAD 2

```
enqueue (Q, v) {
  Node_t node = malloc(…);
  node->val = v;
  node->next = NULL;
  acquire(lock);
  if (Q->tail)
    Q->tail->next = node;
  else
    Q->head = node;
  Q->tail = node;
  release(lock);
}
```

```
begin-transaction
…
enqueue (Q, v); //no locks
…
end-transaction
```

Transactional Memory

- A transaction is executed atomically: ALL or NONE

- If there is a data conflict between two transactions, only one of them completes; the other is rolled back
  - Both write to the same location
  - One reads from the location another writes

ISA-level Tradeoff: Supporting TM

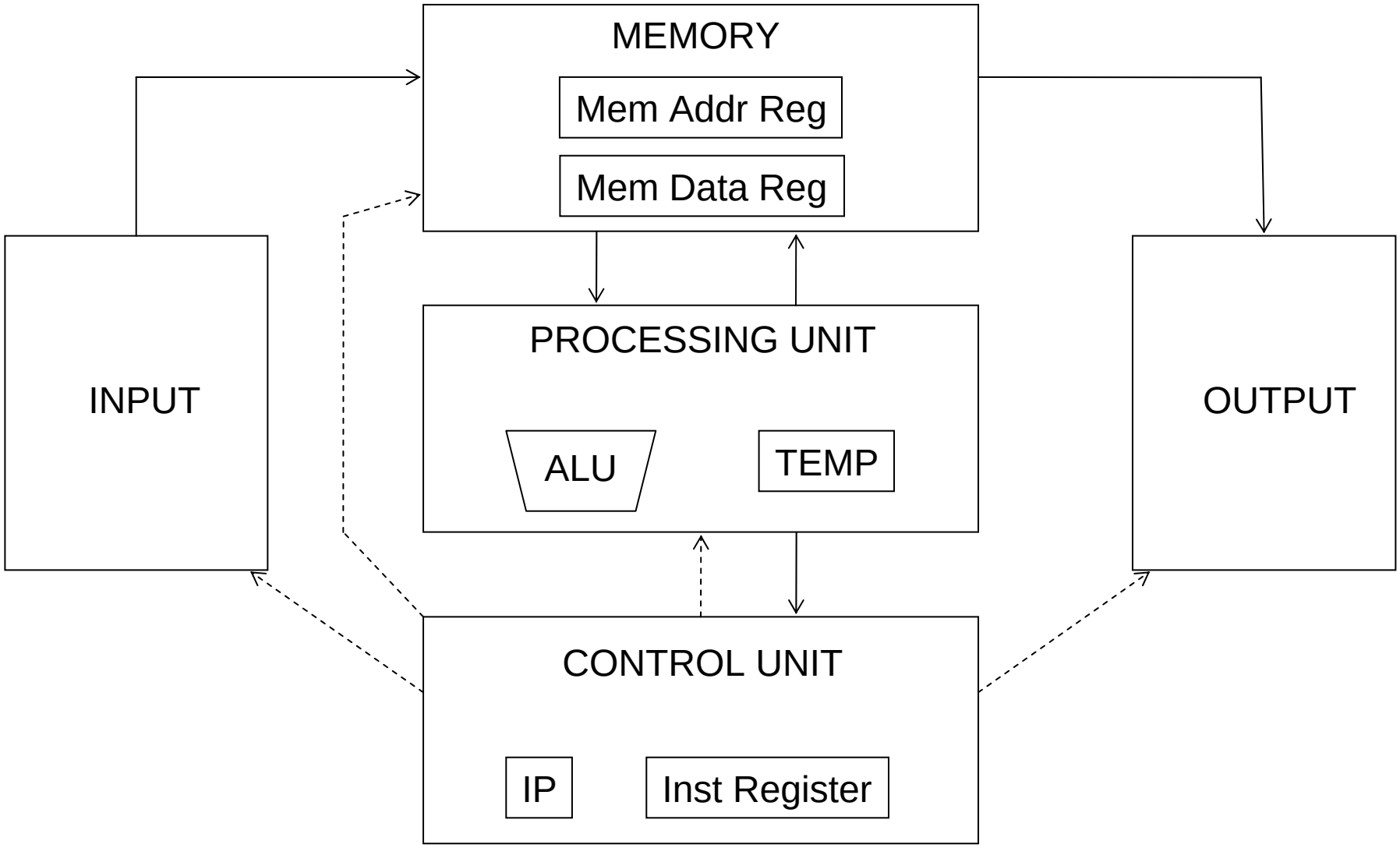
- Still under research
- Pros:
  - Could make programming with threads easier
  - Could improve parallel program performance vs. locks. Why?
- Cons:
  - What if it does not pan out?
  - All future microarchitectures might have to support the new instructions (for backward compatibility reasons)
  - Complexity?
- How does the architect decide whether or not to support TM in the ISA? (How to evaluate the whole stack)

ISA-level Tradeoffs: Instruction Pointer

- Do we need an instruction pointer in the ISA?
  - Yes: Control-driven, sequential execution
    - An instruction is executed when the IP points to it
    - IP automatically changes sequentially (except control flow instructions)
  - No: Data-driven, parallel execution
    - An instruction is executed when all its operand values are available (data flow)

- Tradeoffs: MANY high-level ones
  - Ease of programming (for average programmers)?
  - Ease of compilation?
  - Performance: Extraction of parallelism?
  - Hardware complexity?

The Von-Neumann Model

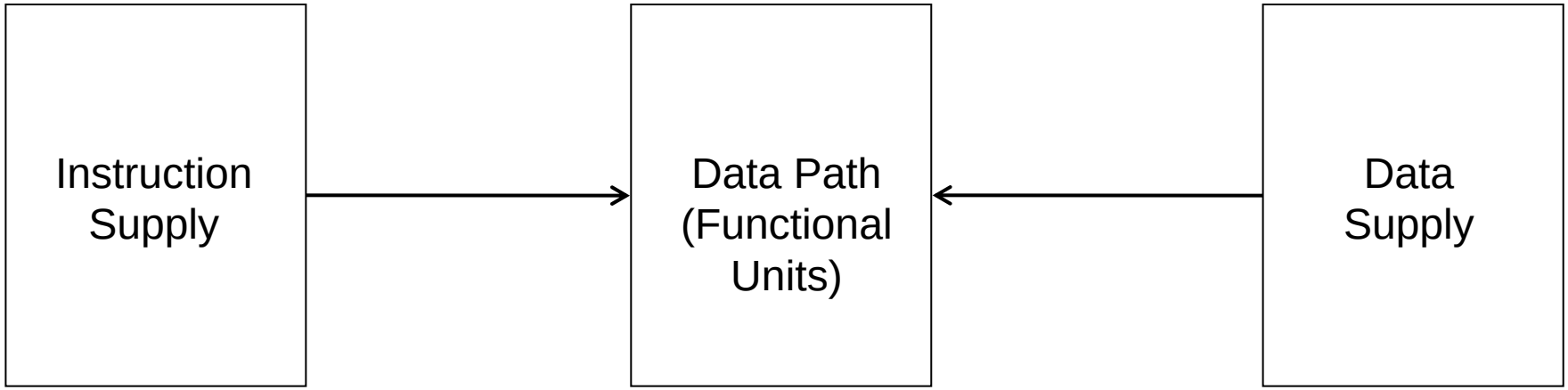


The Von-Neumann Model

- Stored program computer (instructions in memory)
- One instruction at a time
- Sequential execution
- Unified memory
  - The interpretation of a stored value depends on the control signals

- All major ISAs today use this model
- Underneath (at uarch level), the execution model is very different
  - Multiple instructions at a time
  - Out-of-order execution
  - Separate instruction and data caches

Fundamentals of Uarch Performance Tradeoffs



- Zero-cycle latency (no cache miss)
- No branch mispredicts
- No fetch breaks

- Perfect data flow (reg/memory dependencies)
- Zero-cycle interconnect (operand communication)
- Enough functional units
- Zero latency compute?

- Zero-cycle latency
- Infinite capacity
- Zero cost

We will examine all these throughout the course (especially data supply)

How to Evaluate Performance Tradeoffs

$$\text{Execution time} = \frac{\text{time}}{\text{program}}$$

$$= \frac{\text{\# instructions}}{\text{program}} \times \frac{\text{\# cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}}$$

Algorithm	ISA	Microarchitecture
Program	Microarchitecture	Logic design
ISA		Circuit implementation
Compiler		Technology

Improving Performance

- Reducing instructions/program
- Reducing cycles/instruction (CPI)
- Reducing time/cycle (clock period)

Improving Performance (Reducing Exec Time)

- Reducing instructions/program
  - More efficient algorithms and programs
  - Better ISA?

- Reducing cycles/instruction (CPI)
  - Better microarchitecture design
    - Execute multiple instructions at the same time
    - Reduce latency of instructions (1-cycle vs. 100-cycle memory access)

- Reducing time/cycle (clock period)
  - Technology scaling
  - Pipelining

Improving Performance: Semantic Gap

- Reducing instructions/program
  - Complex instructions: small code size (+)
  - Simple instructions: large code size (--)

- Reducing cycles/instruction (CPI)
  - Complex instructions: (can) take more cycles to execute (--)
    - REP MOVS
    - How about ADD with condition code setting?
  - Simple instructions: (can) take fewer cycles to execute (+)

- Reducing time/cycle (clock period)
  - Does instruction complexity affect this?
    - It depends