EITF20: Computer Architecture

Part 2.1.1: Instruction Set Architecture

Liang Liu
liang.liu@eit.lth.se

Computer architecture

Computer architecture is a set of disciplines that describe the functionality, organization and implementation of computer systems.

- **ISA: Instruction-set architecture**
- **Computer orginization: micro architecture**
- **Specific implementation**

Outline

- Computers
- Computer Architecture
- This Course
- Trends
- **Performance**
- Quantitative Principles

What is Performance?

Plane	DC to Paris	Speed
Boeing 747	6.5 h	980 km/h
Concorde	3 h	2160 km/h

- **Time to complete a task (T_{exe})**
  - Execution time, response time, latency
- **Task per day, hour…**
  - Total amount of tasks for given time
  - Thoughput, bandwidth
- **Speed of Concorde vs Boeing 747**
- **Throughput of Boeing 747 vs Concorde**

Performance

$$Performance(X) = \frac{1}{T_{exe}(X)}$$

"X is n times faster than Y" means:

$$\frac{T_{exe}(Y)}{T_{exe}(X)} = \frac{Performance(X)}{Performance(Y)} = n$$

How to define execution time?

Aspect of CPU performance

$$\text{CPUtime} = \text{Execution time} = \\ seconds/program =$$

$$\underbrace{(executed)instr./program}_{IC} * \underbrace{cycles/instr.}_{CPI} * \underbrace{seconds/cycle}_{T_c}$$

	IC	CPI	T_c
Program	X		
Compiler	X	(X)	
Instr. Set	X	X	
Organization		X	X
Technology			X

Instructions are not created equally

"Average Cycles per Instruction"

CPI_{op} = Cycles per Instruction of type *op*

IC_{op} = Number of executed instructions of type *op*

$$CPUtime = T_c * \sum (CPI_{op} * IC_{op})$$

"Instruction frequency"

$$\overline{CPI} = \sum (CPI_{op} * F_{op}) \text{ where } F_{op} = IC_{op}/IC$$

Average CPI: example

Op	F_{op}	CPI_{op}	$F_{op} * CPI_{op}$	% time
ALU	50 %	1	0.5	(33 %)
Load	20 %	2	0.4	(27 %)
Store	10 %	2	0.2	(13 %)
Branch	20 %	2	0.4	(27 %)

$\overline{CPI} = 1.5$

Invest resources where time is spent!

Outline

- Computers
- Computer Architecture
- This Course
- Trends
- Performance
- **Quantitative Principles**

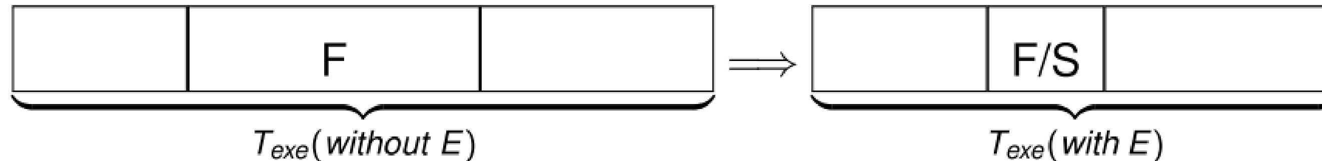
Quantitative Principles

This is intro to design and analysis

- Take advantage of parallelism
  - ILP, DLP, TLP, ...
- Principle of locality
  - 90% of execution time in only 10% of the code
- Focus on the common case
  - In making a design trade-off, favor the frequent case ove the infrequent case
- Amdahl’s Law
  - The performance improvement gained from uisng faster mode is limited by the fraction of the time the faster mode can be used
- The Processor Performance Equation

Amdahl's Law

Enhancement E accelerates a fraction F of a program by a factor S



Speedup due to enhancement E:

$$Speedup(E) = \frac{T_{exe}(without\ E)}{T_{exe}(with\ E)} = \frac{Performance(with\ E)}{Performance(without\ E)}$$

$$T_{exe}(with\ E) = T_{exe}(without\ E) * [(1 - F) + F/S]$$

$$Speedup(E) = \frac{T_{exe}(without\ E)}{T_{exe}(with\ E)} = \frac{1}{(1-F)+F/S}$$

Best you could ever hope to do:

$$Speedup_{maximum} = \frac{1}{(1 - Fraction_{enhanced})}$$

Amdahl's Law: example

- New CPU is 10 times faster!
- 60% for I/O which remains almost the same…

$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{enhanced}}\right) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

$$= \frac{1}{(1-0.4) + \frac{0.4}{10}} = \frac{1}{0.64} = 1.56$$

Apparently, its human nature to be attracted by 10X faster, vs. keeping in perspective its just 1.6X faster

Outline

- Reiteration
- **Instruction Set Principles**
- The Role of Compilers
- MIPS

Instruction Set

- Serves as an interface between software and hardware
- Provides a mechanism by which the software tells the hardware what should be done
- Basic functionality that hardware can provide to software

High level language code : C, C++, Java, Fortran,

↓ compiler

Assembly language code: architecture specific statements

↓ assembler

Machine language code: architecture specific bit patterns

software

instruction set

hardware

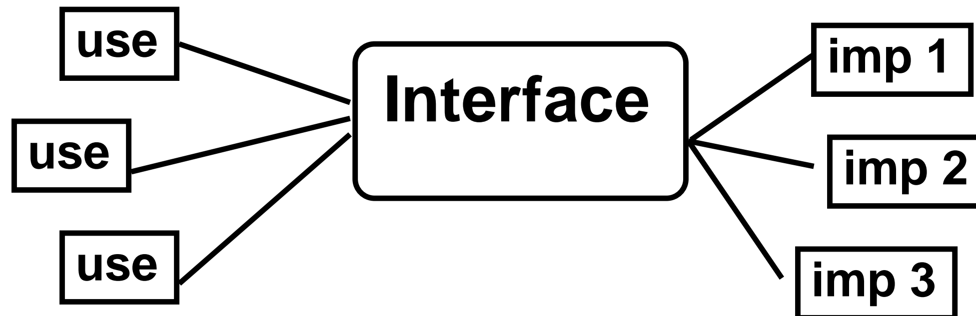
Interface Design

- A good interface?

Interface Design

A good interface

- Lasts through many implementations (portability, compatibility)
- Can be used in many ways (generality)
- Provides sufficient functionality to higher levels
- Permits an efficient implementation at lower levels
- …



ISA Classification

What's needed in an instruction set?

- Addressing
- Operands
- Operations
- Control Flow

ISA Classification

- **Where are operands stored?**
  - registers, memory, stack, accumulator
- **How many explicit operands are there?**
  - 0, 1, 2, or 3
- **How is the operand location specified?**
  - register, immediate, indirect, . . .
- **What type & size of operands are supported?**
  - byte, int, float, double, string, vector. . .
- **What operations are supported?**
  - add, sub, mul, move, compare . . .
- **How is the operation flow controlled?**
  - branches, jumps, procedure calls . . .
- **What is the encoding format**
  - fixed, variable, hybrid...

ISA Classes: Where are operands stored

(a) Stack
(b) Accumulator
(c) Register-memory
(d) Register-register/load-store

Processor

TOS

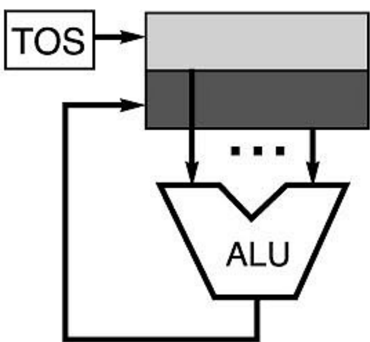
ALU ALU ALU ALU

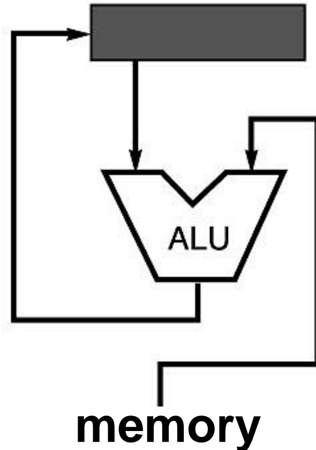
Memory

1960s to 1970s | before 1960 | 1970s to present | 1960s to present

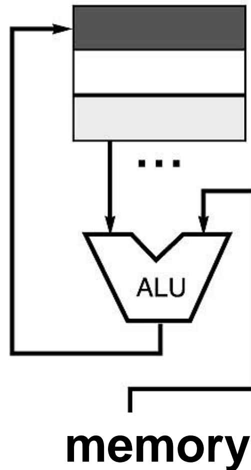
Example: C=A+B

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1, A Load R2, B Add R3, R1, R2 Store C, R3

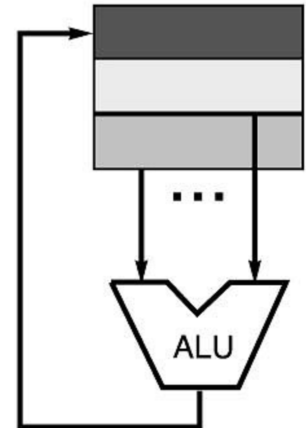




acc = acc + mem[B]



R1 = R1 + mem[B]

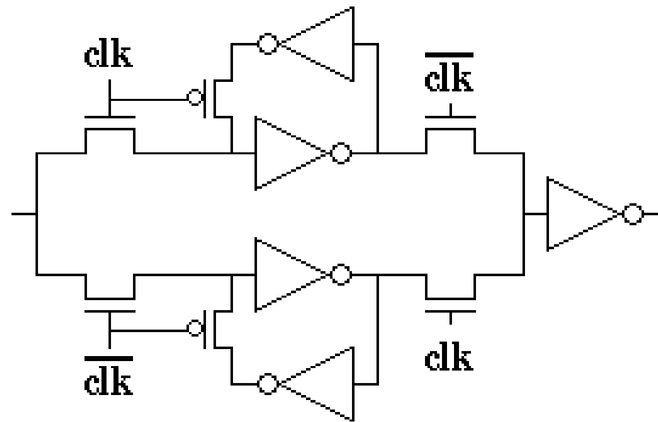


R3 = R1 + R2

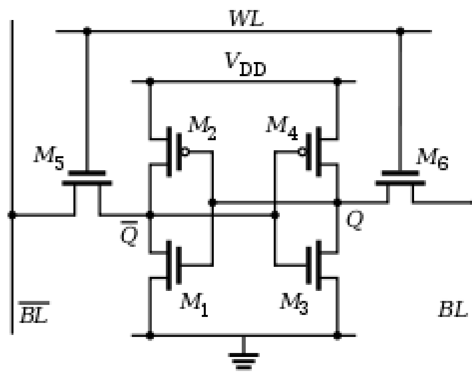
GPR (General Purpose Register)

- **Registers are much faster than memory (even cache)**
  - Register values are available “immediately”
  - When memory isn’t ready, processor must wait (“stall”)
- **Registers are convenient for variable storage**
  - Compiler assigns some variables (especially frequently used ones) just to registers
  - More compact instr. since small fields specify registers (compared to memory addresses)
- **Disadvantages**
  - Higher instruction count (load/store)
  - Dependent on good compiler (Reg. assignment)
  - Higher hardware cost (comparing to MEM)

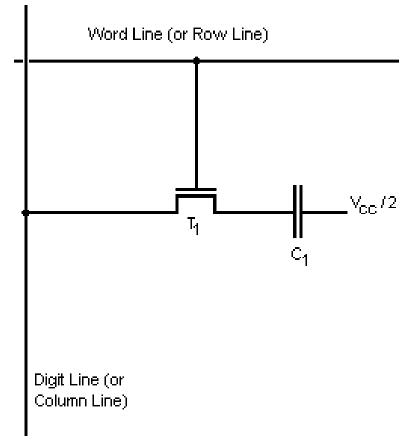
Register, SRAM, DRAM



Register (DFF) Cell (16T)

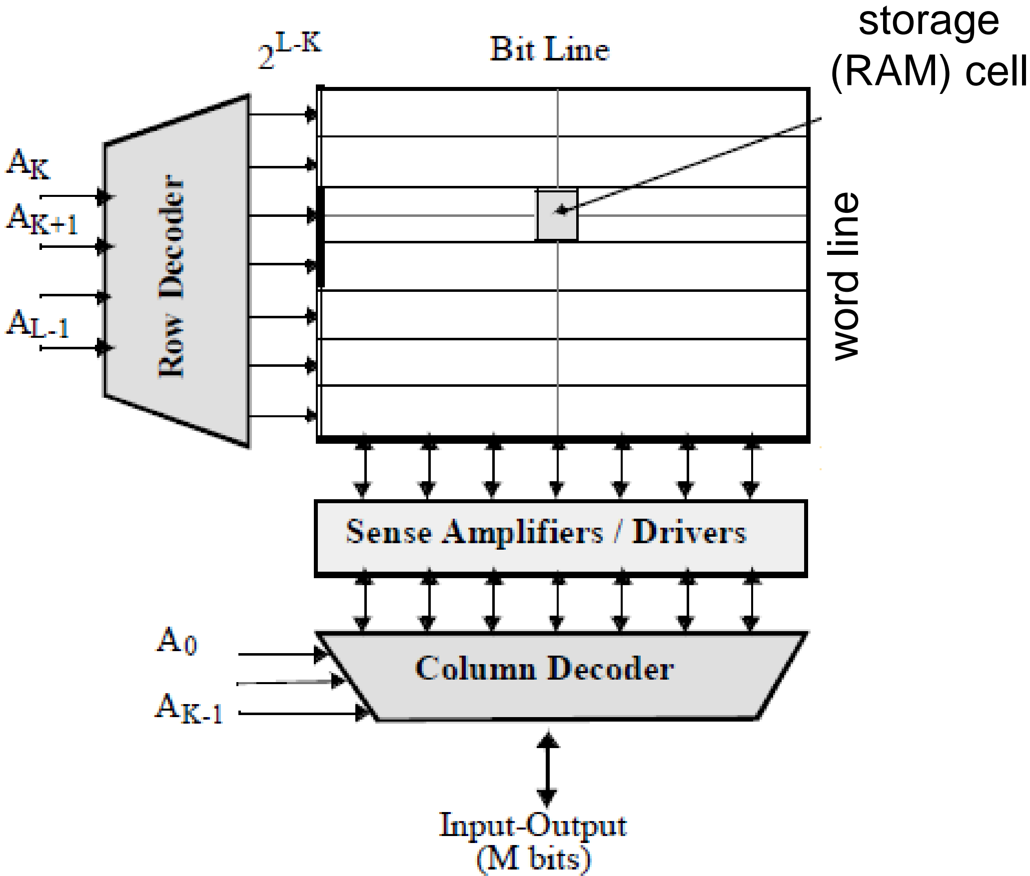


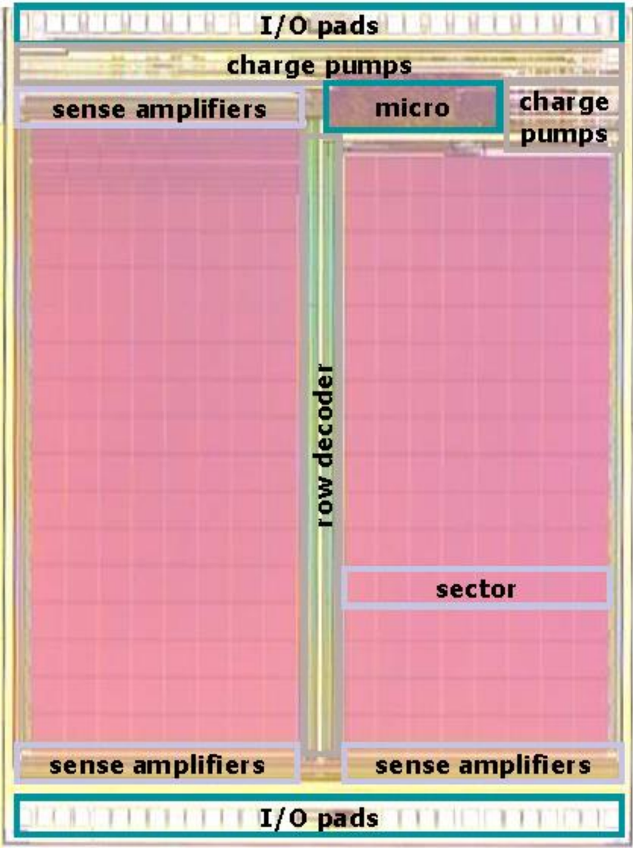
SRAM Cell (6T)



DRAM Cell (1T)

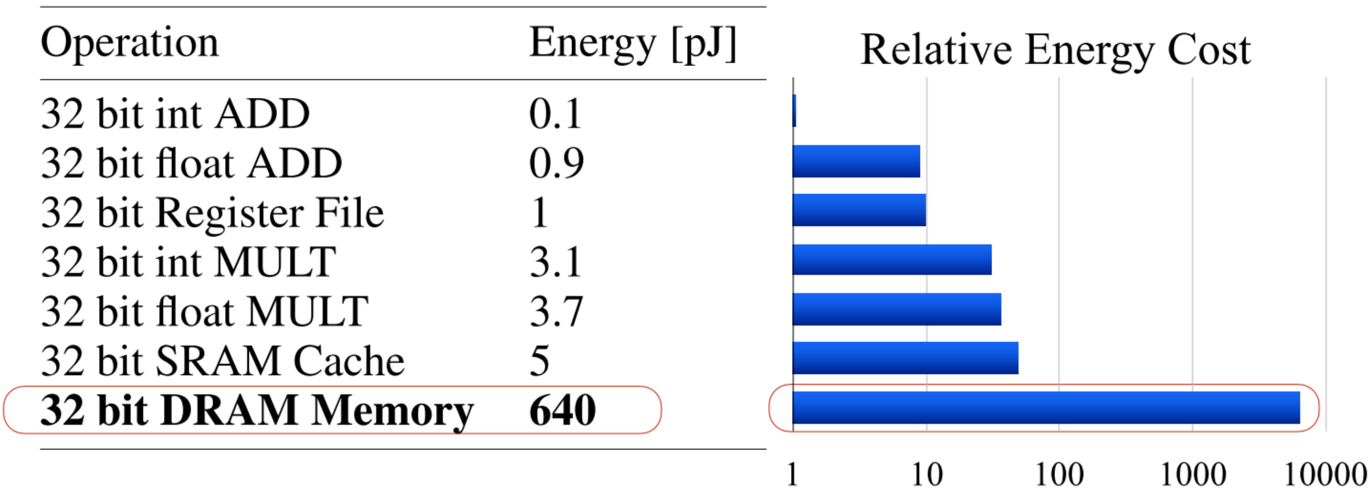
Memory Architecture





Reg v.s. Mem (65nm CMOS)

	Register Bank	Memory
Size	256*4Byte	1K*4Byte
Area	$0.14mm^2$	$0.04mm^2$
Density	$7KB/mm^2$	$100KB/mm^2$

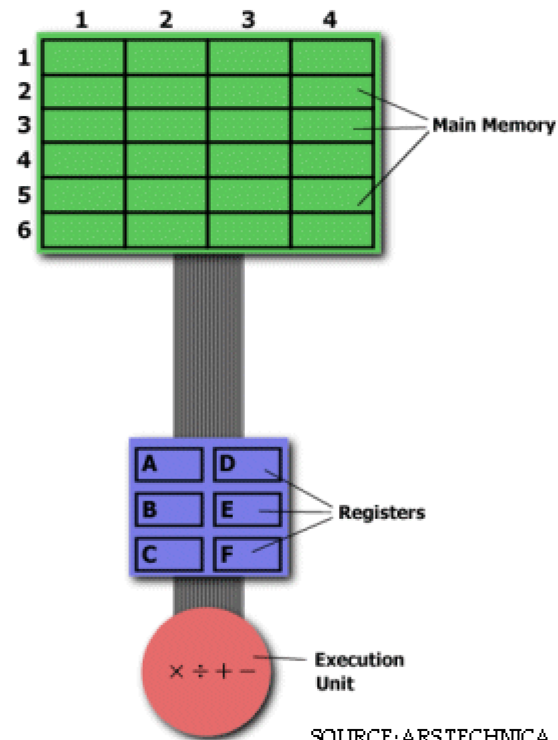


Operation	Energy [pJ]
32 bit int ADD	0.1
32 bit float ADD	0.9
32 bit Register File	1
32 bit int MULT	3.1
32 bit float MULT	3.7
32 bit SRAM Cache	5
32 bit DRAM Memory	**640**

Example: RISC-CICS

```
MULT 2:3, 5:2
```

```
LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A
```



CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	"Single"-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes Less memory access (instr.)	large code sizes
Irregular Instruction size	Regular Instruction size

Example: RISC-CICS

Execution Resources

Micro-Operations Buffer

Micro-Op Dispatch

Fetch

Decode & Rename Registers

Commit & Retire

$$\text{CPUtime} = \text{Execution time} = \\ \textit{seconds}/\textit{program} = \\ \underbrace{(\textit{executed})\textit{instr.}/\textit{program}}_{IC} * \underbrace{\textit{cycles}/\textit{instr.}}_{CPI} * \underbrace{\textit{seconds}/\textit{cycle}}_{T_c}$$

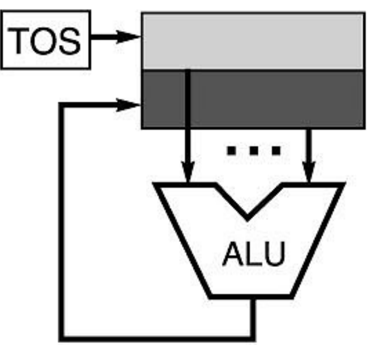
IoT Processor?

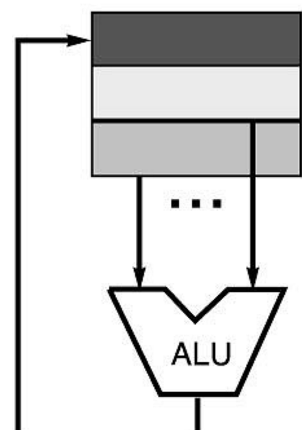
Stack	Register (load-store)
Push A Push B Add Pop C	Load R1, A Load R2, B Add R3, R1, R2 Store C, R3

- **Advantages**
  - Very compact object code → small memory
  - Simple compilers (no reg. assignment)
  - Fast operand access (no addressing)

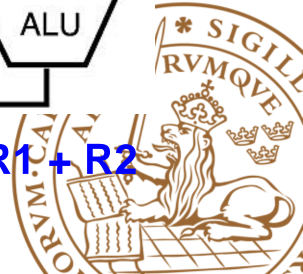
    → More efficient instruction
  - Minimal processor state

    → simple hardware, e.g., instruction decoder



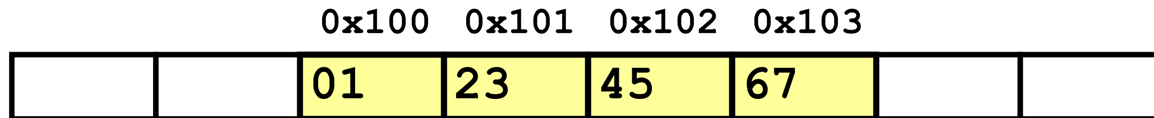


R3 = R1 + R2

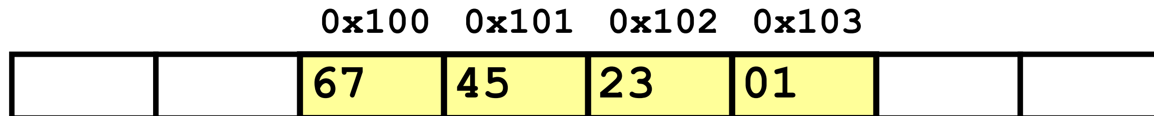


Memory Addressing

- **A 32-bit (4Byte) integer variable (0x01234567) stored at address 0x100**
  - Big Endian
    - **Least significant byte has highest address**



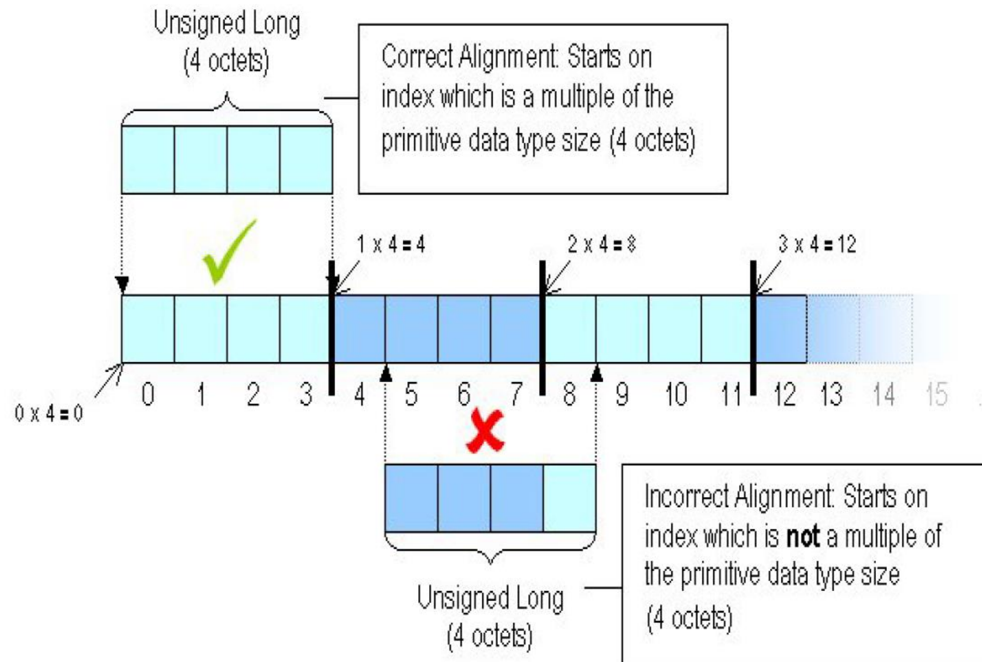
  - Little Endian
    - **Least significant byte has lowest address**



  - **Important for exchange of data**

Memory Addressing

- **Memory is generally byte addressed and provides access for**
  - bytes (8 bits), half words (16 bits), words (32 bits), and double words(64 bits)
- **An architecture may require that data is aligned:**
  - Address index is multiple of date type size (depending on memory implementation)



Memory Addressing Mode

Addressing Mode	Example	Action
1. Register direct	Add R4, R3	R4 <- R4 + R3
2. Immediate	Add R4, #3	R4 <- R4 + 3
3. Displacement	Add R4, 100(R1)	R4 <- R4 + M[100 + R1]
4. Register indirect	Add R4, (R1)	R4 <- R4 + M[R1]
5. Indexed	Add R4, (R1 + R2)	R4 <- R4 + M[R1 + R2]
6. Direct	Add R4, (1000)	R4 <- R4 + M[1000]
7. Memory Indirect	Add R4, @(R3)	R4 <- R4 + M[M[R3]]
8. Auto-increment	Add R4, (R2)+	R4 <- R4 + M[R2] R2 <- R2 + d
9. Auto-decrement	Add R4, (R2)-	R4 <- R4 + M[R2] R2 <- R2 - d
10. Scaled	Add R4, 100(R2)[R3]	R4 <- R4 + M[100 + R2 + R3*d]

ISA Classification

What's needed in an instruction set?

- Addressing
- **Operands**
- Operations
- Control Flow

What does it mean?

00100000010010000110100100100001

What does it mean?

00100000010010000110100100100001

001000	00010	01000	0110100100100001
instruction	source	target	immediate
ADDI	2	8	26913

ADDI R8,R2,26913

Types and sizes of operands

- integer
- floating point (single precision)
- character
- packed decimal
- ... etc ...

Floating point v.s. fixed point

	32, 64bit Fixed-point (with 2, 4 pipeline stages)		32, 64bit Floating-point (with 6, 8 pipeline stages)		32, 64bit Floating-point (with 18, 23 pipeline stages)	
Area (slices)	36	139	293	693	504	1383
Max Freq. (MHz) achievable	250	250	140	130	250	200
Power (mW) at 100MHz	23.48	104	148.7	329	-	-

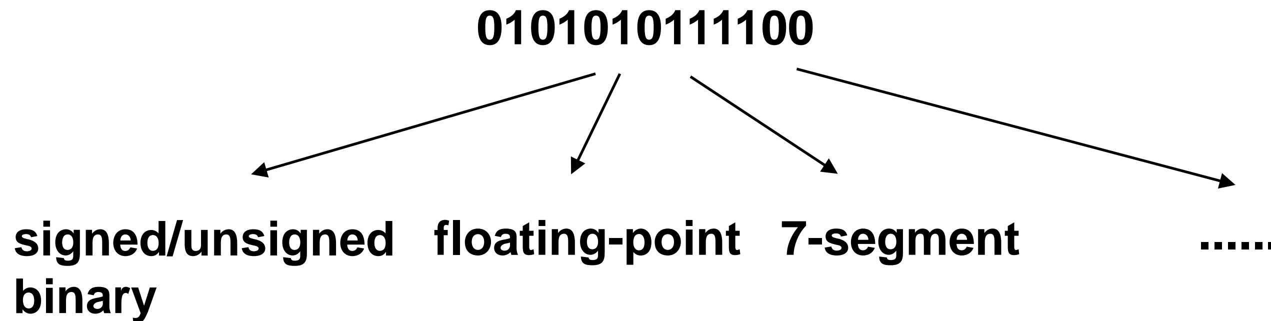
Table 1a: A comparison of addition units (Virtex2Pro-c2vp125-7)

	32, 64bit Fixed-point (with 5, 7 pipeline stages)		32, 64bit Floating-point (with 9, 11 pipeline stages)		64bit Floating-point (with 18, 23 pipeline stages)	
Area (slices)/Embedded multipliers	190 / 4	1024 / 16	249 / 3	775 / 10	492 / 3	1558 / 10
Max Freq. (MHz) achievable	200	130	140	130	200	200
Power (mW) at 100MHz	136.3	804	164.7	424	-	-

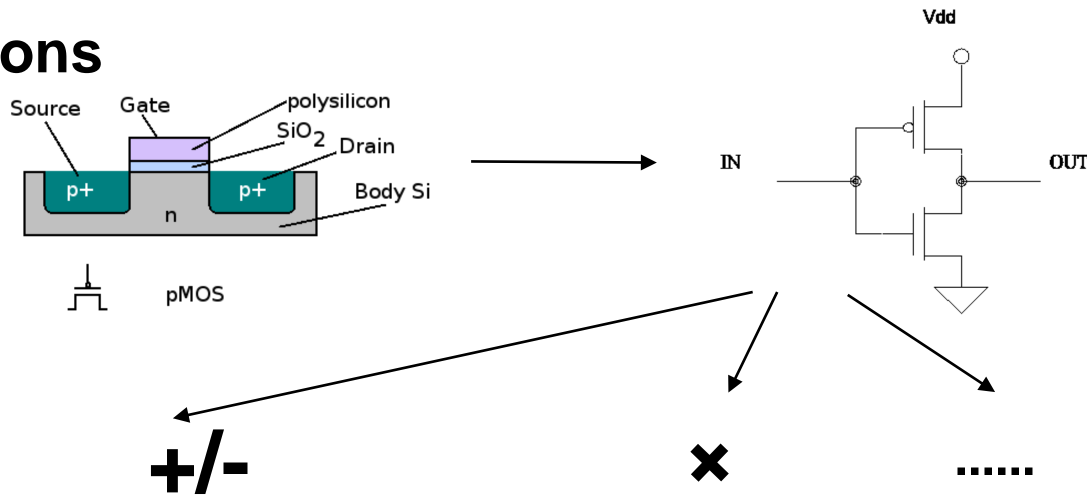
Table 1b: A comparison of multiplication units (Virtex2Pro-c2vp125-7)

Two basic components

- Operands (Data type)



- Operations



- Check what is in the ISA!

ISA Classification

What’s needed in an instruction set?

- Addressing
- Operands
- **Operations**
- Control Flow

Types of operations

• Arithmetic and Logic:	AND, ADD
• Data Transfer:	MOVE, LOAD, STORE
• Control	BRANCH, JUMP, CALL
• System	OS CALL,
• Floating Point	ADDF, MULF, DIVF
• Decimal	ADDD, CONVERT
• String	MOVE, COMPARE, SEARCH
• Graphics	(DE)COMPRESS

Types of operations (frequency)

Rank	*Instruction*	*Frequency*
1	load	22%
2	branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	register move	4%
9	call	1%
10	return	1%
Total		96%

80x86 Instruction Frequency

ISA Classification

- **What's needed in an instruction set?**
  - Addressing
  - Operands
  - Operations
  - **Control Flow**

Types of control instructions

- Conditional branches
- Unconditional branches (jumps)
- Procedure call/returns

	Floating-point average	Integer average
Call/return	8%	19%
Jump	10%	6%
Conditional branch	82%	75%

Frequency of branch instructions (0%, 25%, 50%, 75%, 100%)

© 2007 Elsevier, Inc. All rights reserved.

Instruction format

Variable instruction format

- Compact code but the instruction decoding is more complex and thus slower
- Examples: VAX, Intel 80x86 (1-17 byte)

Operation # operands	Address specifier 1	Address field 1	...	Address specifier x	Address field x

Fixed instruction format

- Easy and fast to decode but gives large code size
- Examples: Alpha, ARM, MIPS (4byte), PowerPC, SPARC

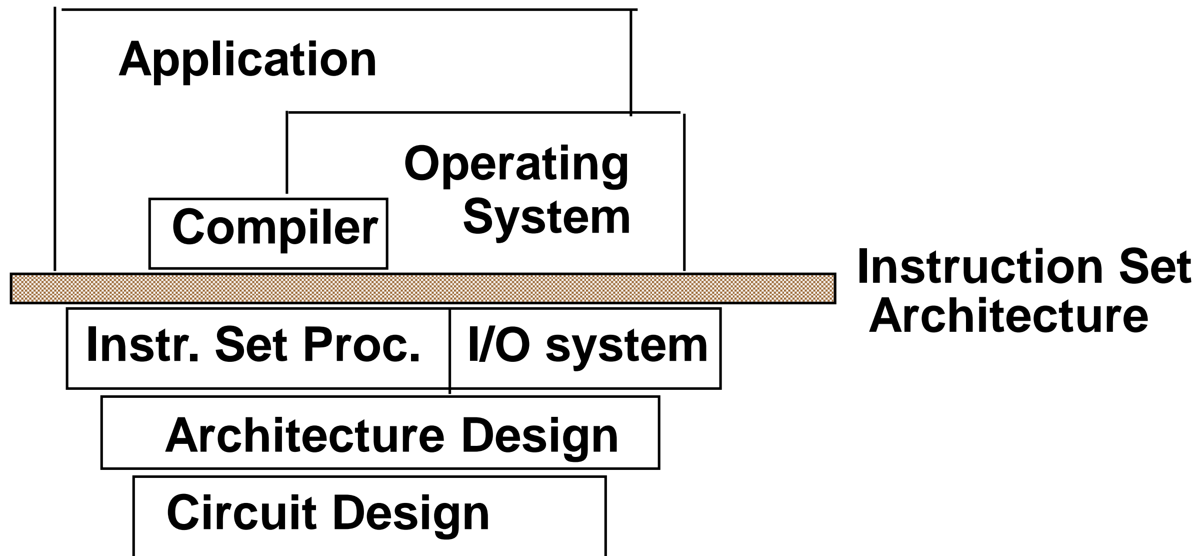
Operation	Address field 1	Address field 2	Address field 3

Outline

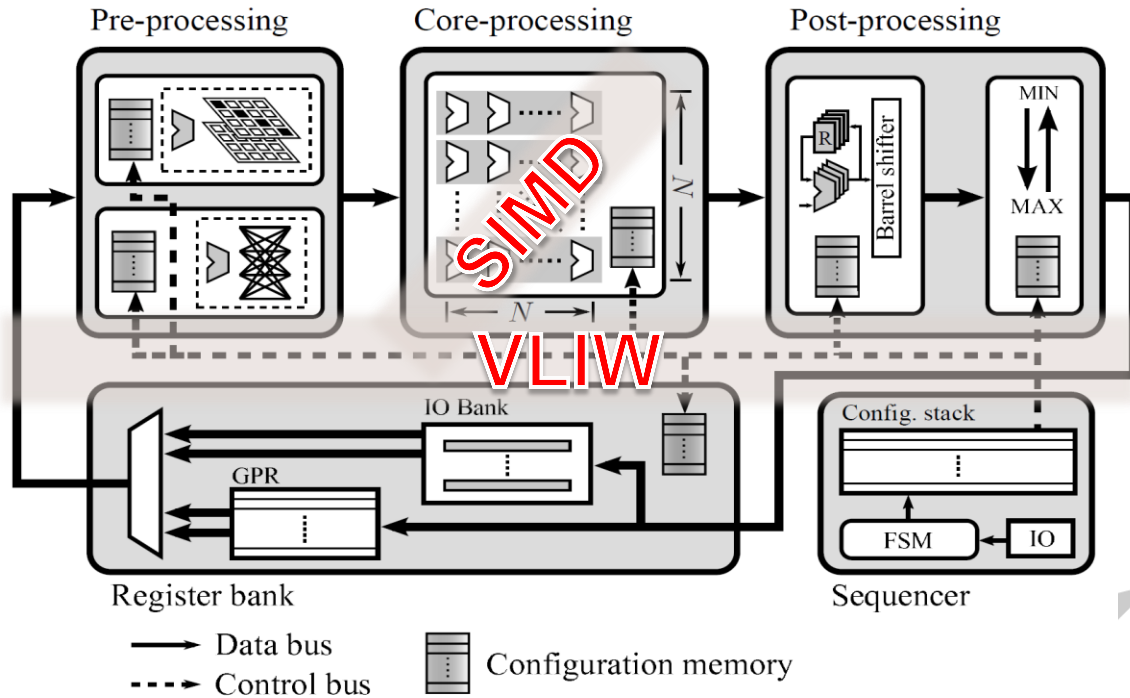
- Reiteration
- Instruction Set Principles
- **The Role of Compilers**
- MIPS

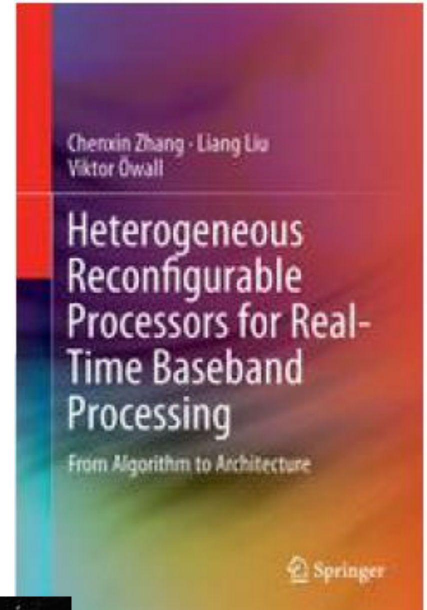
ISA and compiler

- Instruction set architecture is a compiler target
- By far MOST instructions executed are generated by a compiler (exception certain special purpose processors)
- Interaction compiler - ISA critical for overall performance



ISA and compiler: a “good or bad?” example





ISA and compiler: a “good or bad?” example

A	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
	VDPU						VCPU				vmacArray					VDEU				
21			src	vArrayB	1				0					0					0	
20					0				0					0		Reserved			0	
19		Row 0			0				0		Reserved			0					0	
18			opcode	Neg, i	1		Reserved		0					0					0	
17					0				0					0			en_r3	Disable	0	
16			src	vArrayB	1				0			opcode	Complex	0			en_r2	Disable	0	
15					0				0		VDDU			0			en_r1	Disable	0	
14		Row 1			0				0			swap_b	Disable	0			en_r0	Disable	0	
13			opcode	Neg, i	1		perm_en	Enable	1			swap_a	Disable	0		Acc	reg_acc_r3	Direct	0	
12	Process prologue				0			Row 0	0			mul_en	Enable	1			reg_acc_r2	Direct	0	
11			src	vArrayB	1				0			mul_sign_out	Unsigned	1			reg_acc_r1	Direct	0	
10					0			Row 0	0			add_en_l1	Enable	1			reg_acc_r0	Direct	0	
9		Row 2			0		perm_imm		0			add_en_l2	Enable	1			mux_in_r23	Straight	0	
8			opcode	Neg, i	1			Row 3	1			add_sign_l1	Signed	0		Sum	mux_in_r01	Straight	0	
7					0				1		VMAC array	add_sign_l2	Signed	0			mux_out_r23	acc out	0	
6			src	vArrayB	1			Row 3	1			add_ctrl_l1a	Sub	1			mux_out_r01	acc out	0	
5					0				1			add_ctrl_l1b	Add	0			opcode	ASR	0	
4		Row 3			0		swap_en	Disable	0			add_ctrl_l2a	Sub	1					1	
3			opcode	Neg, i	1				0			add_ctrl_l2b	Sub	1		Barral shift			0	
2					0		opcode	None	0			op_coeff	Const mul	0			shift	1	0	
1	Mask		src	None	0				0					1					0	
0					0		mask_en	Disable	0			coeff_shape	Column	0					1	
Inst [HEX]	00252948						000021E0				00001E5A					00000011				

Synopsys ASIP Designer

User-defined architecture

Processor model
nML

Instruction set

FMT	ALU	OPD
FMT	MPY	OPD
FMT	OPD	SH

User-defined algorithm

Algorithm
C/C++

Architectural optimization and software development

Optimizing C/C++ compiler

Asm

Link

Binary

Debugger & profiler

Instruction set simulator

Refinement

ASIP synthesis

RTL generator

Synthesizable RTL
VHDL/Verilog

VCS

Design Compiler

Virtual prototype

Verification

Test program generator

ASIC
FPGA

1. SDK generation
2. Architectural optimization
3. Hardware generation
4. Verification

The role of compilers

High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

Machine Interpretation

Control Signal Specification

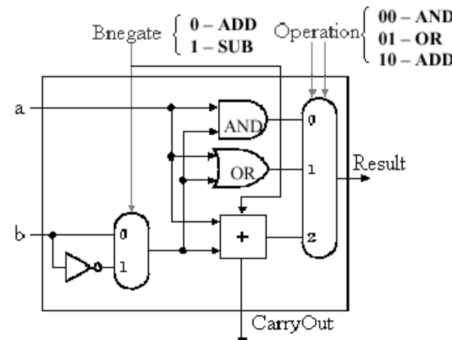
temp = v[k];

v[k] = v[k+1];

v[k+1] = temp;

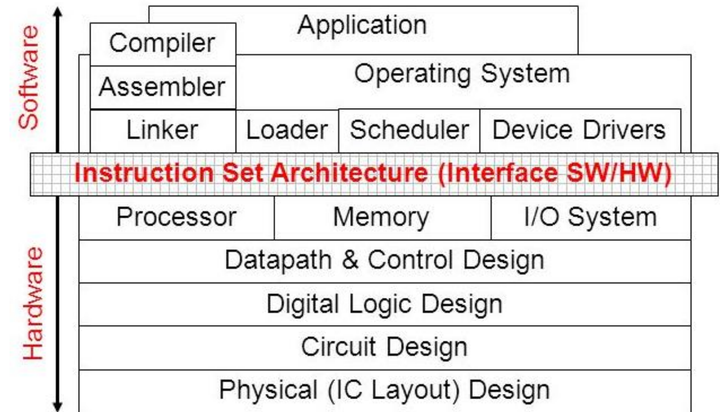
```
lw  $15,    0($2)
lw  $16,    4($2)
sw  $16,    0($2)
sw  $15,    4($2)
```

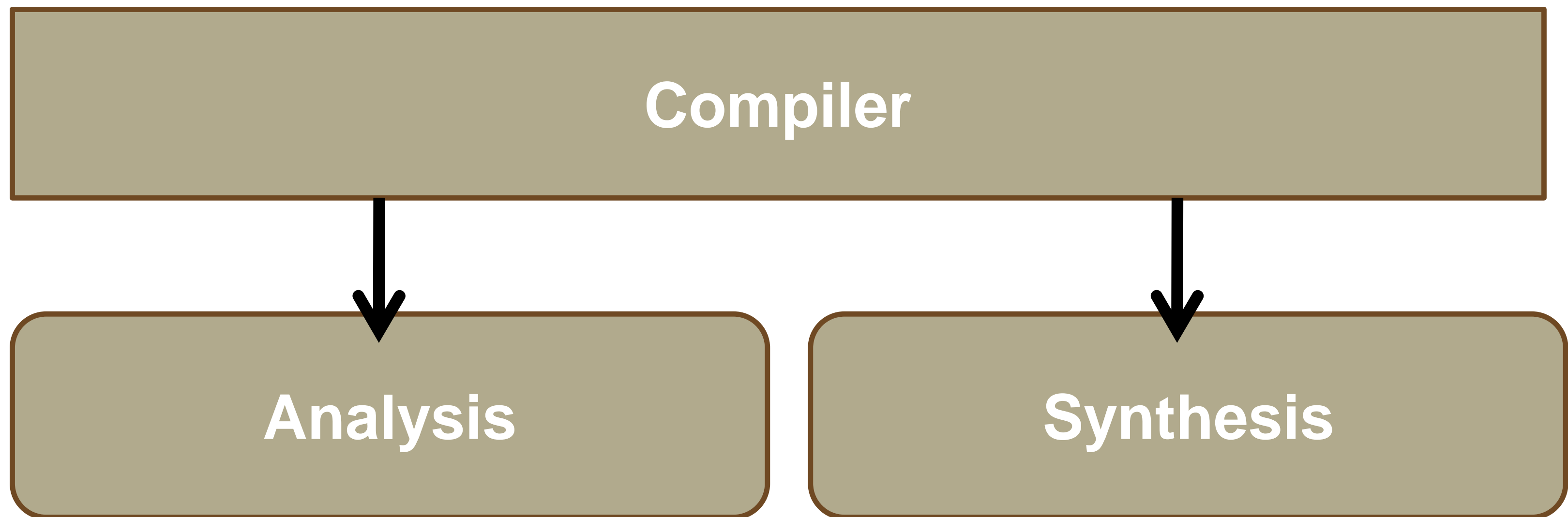
```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```



The structure of a compiler

- **Any compiler must perform two major tasks**
  - Analysis of the source program
  - Synthesis of a machine-language program





Example of compiler optimization

- **Code improvements made by the compiler are called optimizations and can be classified:**
  - High-order transformations: procedure inlining
  - Optimizations: dead code elimination
  - Constant propagation
  - Common sub-expression elimination
  - Loop-unrolling
  - Register allocation (almost most important)
  - Machine-dependent optimizations (takes advantage of specific architectural features)

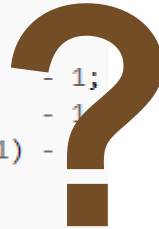
Example of compiler optimization

```
int pred(int x) {
    if (x == 0)
       return 0;
    else
       return x - 1;
}
```

Procedure inlining

```
int f(int y) {
    return pred(y) + pred(0) + pred(y+1);
}
```

```
int f(int y) {
    int temp;
    if (y   == 0) temp  = 0; else temp  = y       - 1;
    if (0   == 0) temp += 0; else temp += 0       - 1
    if (y+1 == 0) temp += 0; else temp += (y + 1) -
    return temp;
}
```



```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

```
int x = 14;
int y = 0;
return 0;
```

Constant propagation

```
int foo(void)
{
  int a = 24;
  int b = 25; /* Assignment to dead variable */
  int c;
  c = a << 2;
  return c;
  b = 24; /* Unreachable code */
  return 0;
}
```

Dead code elimination

```
a = b * c + g;
d = b * c * e;
```

```
tmp = b * c;
a = tmp + g;
d = tmp * e;
```

Common expression elimination

Example of compiler optimization

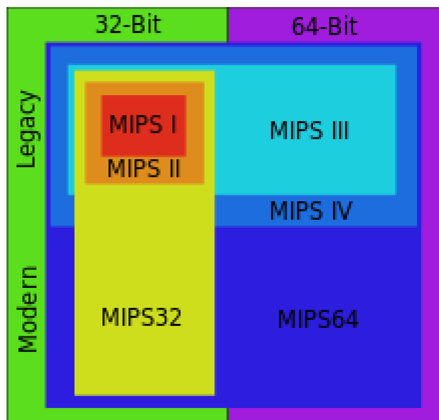
- **Code improvements made by the compiler are called optimizations and can be classified:**
  - High-order transformations: procedure inlining
  - Optimizations: dead code elimination
  - Constant propagation
  - Common sub-expression elimination
  - Loop-unrolling
  - Register allocation (almost most important)
  - Machine-dependent optimizations (takes advantage of specific architectural features)
- **Almost all these optimizations are easier to do if there are many general registers available!**
  - E.g., common sub/expression elimination stores temporary value into a register
  - Loop-unrolling
  - Procedure inlining

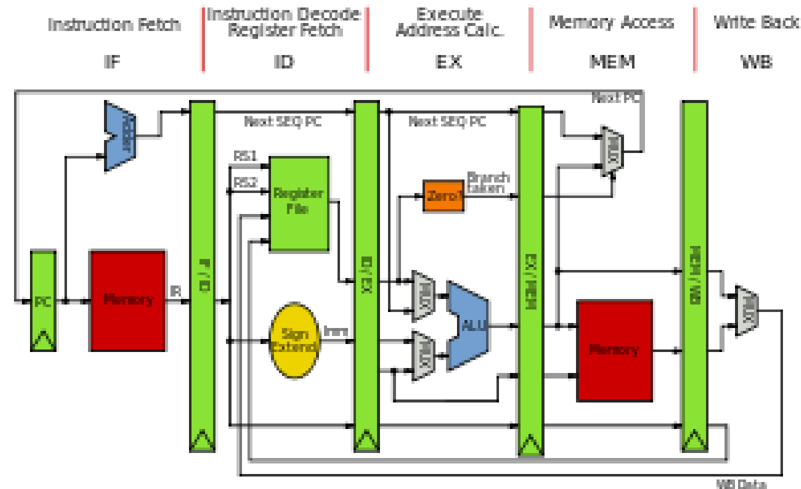
Outline

- Reiteration
- Instruction Set Principles
- The Role of Compilers
- **MIPS**

The MIPS64 architecture

- **An architecture representative of modern ISA:**
  - 64-bit load/store GPR architecture
  - 32 general integer registers (R0 = 0) and 32 floating point registers
  - Supported data types: bytes, half word (16 bits), word (32 bits), double word (64 bits), single and double precision IEEE floating points
  - Memory byte addressable with 64-bit addresses
  - Addressing modes: immediate and displacement





MIPS instruction example

LW	R1,60(R7)	Load word
SB	R2,41(R5)	Store byte
MUL	R2,R1,R3	Integer multiply
AND	R3,R2,R1	Logical AND
DADDI	R5,R6,#17	Add immediate
J	lable	Jump
BEQZ	R4,lable	Branch if R4 zero
JALR	R7	Procedure call

MIPS instruction format

I-type instruction

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction

Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

J-type instruction

Jump and jump and link
Trap and return from exception

© 2007 Elsevier, Inc. All rights reserved

Summary

- The instruction set architecture has importance for the performance
- The important aspects of an ISA are:
  - register model
  - addressing modes
  - types of operations
  - data types
  - encoding
- Benchmark measurements can reveal the most common case
- Interaction compiler