50.002 Computation Structures
Software Abstraction & Assembly Language

Oliver Weeger

2018 Term 3, Week 5, Session 1

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN
INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

β machine language: 32-bit instructions

3 registers:

OPCODE	r_c	r_a	r_b	*unused*

arithmetic: ADD, SUB, MUL, DIV
compare: CMPEQ, CMPLT, CMPLE
boolean: AND, OR, XOR
shift: SHL, SHR, SRA

Ra and Rb are the operands,
Rc is the destination.
R31 reads as 0, unchanged by writes

2 registers, 1 const:

OPCODE	r_c	r_a	16-bit signed constant

arithmetic: ADDC, SUBC, MULC, DIVC
compare: CMPEQC, CMPLTC, CMPLEC
boolean: ANDC, ORC, XORC
shift: SHLC, SHRC, SRAC
branch: BNE/BT, BEQ/BF (const = word displacement from PC_{NEXT})
jump: JMP (const not used)
memory access: LD, ST (const = byte offset from Reg[ra])

Two's complement 16-bit constant for
numbers from −32768 to 32767;
sign-extended to 32 bits before use.

6-bit OPCODES:

2:0 / 5:3	000	001	010	011	100	101	110	111
000								
001								
010								
011	LD	ST		JMP		BEQ	BNE	LDR
100	ADD	SUB	MUL*	DIV*	CMPEQ	CMPLT	CMPLE	
101	AND	OR	XOR		SHL	SHR	SRA	
110	ADDC	SUBC	MULC*	DIVC*	CMPEQC	CMPLTC	CMPLEC	
111	ANDC	ORC	XORC		SHLC	SHRC	SRAC	

β [Mem] instructions: LD & ST

- Load: "Load into $\mathbf{r_c}$ the contents of the memory location whose address is the content of $\mathbf{r_a}$ plus **CONST**"
- $Reg[\mathbf{r_c}] \leftarrow Mem[\ Reg[\mathbf{r_a}] + sxt(\mathbf{CONST})\ ]$
- `LD(ra,Const,rc)`
  or `LD(Const,rc)=LD(R31,Const,rc)`

6 bits	5 bits	5 bits	16 bits
OPCODE	$\mathbf{r_c}$	$\mathbf{r_a}$	**16-bit signed CONST**

- Store: "Store the contents of $\mathbf{r_c}$ into the memory location whose address is the content of $\mathbf{r_a}$ plus **CONST**"
- $Mem[\ Reg[\mathbf{r_a}] + sxt(\mathbf{CONST})\ ] \leftarrow Reg[\mathbf{r_c}]$
- `ST(rc,Const,ra)`
  or `ST(rc,Const)=ST(rc,Const,R31)`

BYTE ADDRESSES, but only 32-bit/4-byte word accesses to word-aligned addresses are supported. Low two address bits are ignored!

β [PC] instructions: BEQ, BNE & JMP

- Branch instructions for conditionals: "If r_a is 0 (not 0), save the current location (PC) into r_c and continue at **label** location (add **CONST** to PC)"

- `BEQ(ra,label,rc)` (branch if equal)
  PC = PC + 4;
  Reg[r_c] = PC;
  if (REG[r_a] == 0)
      PC = PC + 4***CONST**

  `BNE(ra,label,rc)` (branch if not equal)
  PC = PC + 4;
  Reg[r_c] = PC;
  if (REG[r_a] != 0)
      PC = PC + 4***CONST**

6 bits	5 bits	5 bits	16 bits
OPCODE	r_c	r_a	**16-bit signed CONST**

CONST = (<u>label</u> - <addr of BNE/BEQ>)/4 – 1
(up to 32767 instructions before/after BNE/BEQ)

- Here, the **<u>label</u>** refers directly to an address, which needs to be converted to the **CONST** that specifies the word offset of the address from the current PC.

- Abbreviations:
  `BEQ(ra,label)=BEQ(ra,label,R31)=BF(…)`
  `BNE(ra,label)=BNE(ra,label,R31)=BT(…)`

- Unconditional branches:
  `BR(label,rc)=BEQ(R31,label,rc)`
  `BR(label)    =BEQ(R31,label,R31)`

Encoding binary instructions

- What we want to do:

“Add the contents of **R1** to the contents of **R2** and store the result in **R3**”

Reg[3] ← Reg[1] + Reg[2]

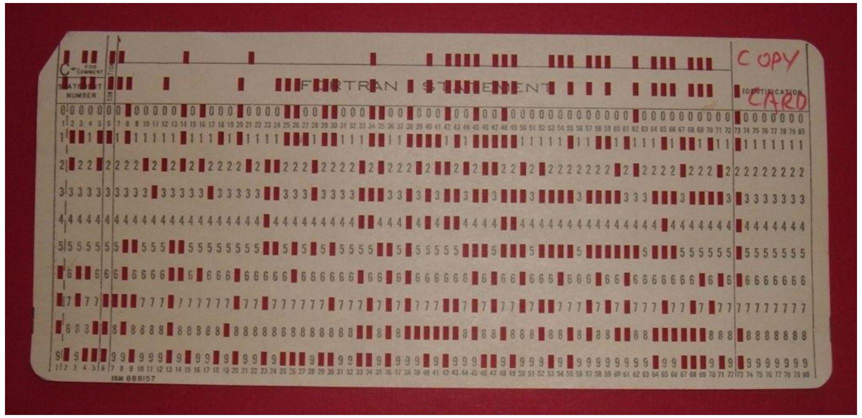
- 32-bit β instruction:

OPCODE	Rc	Ra	Rb	(unused)
100000	00011	00001	00010	00000000000

Assembly

- Assembler language: `ADD(R1,R2,R3)`
- High-level language (C): `c = a+b;`

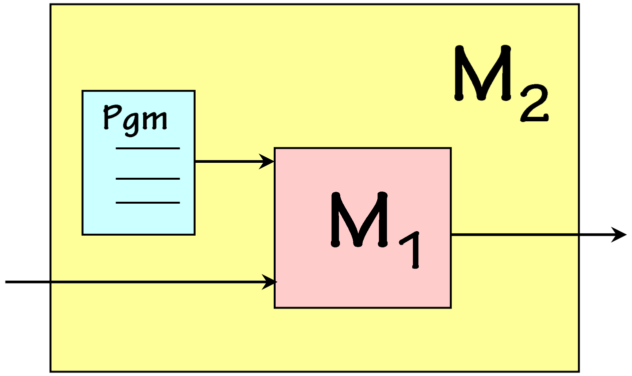
Compilation



IBM 1130 Fortran punched card
https://en.wikipedia.org/wiki/Punched_card

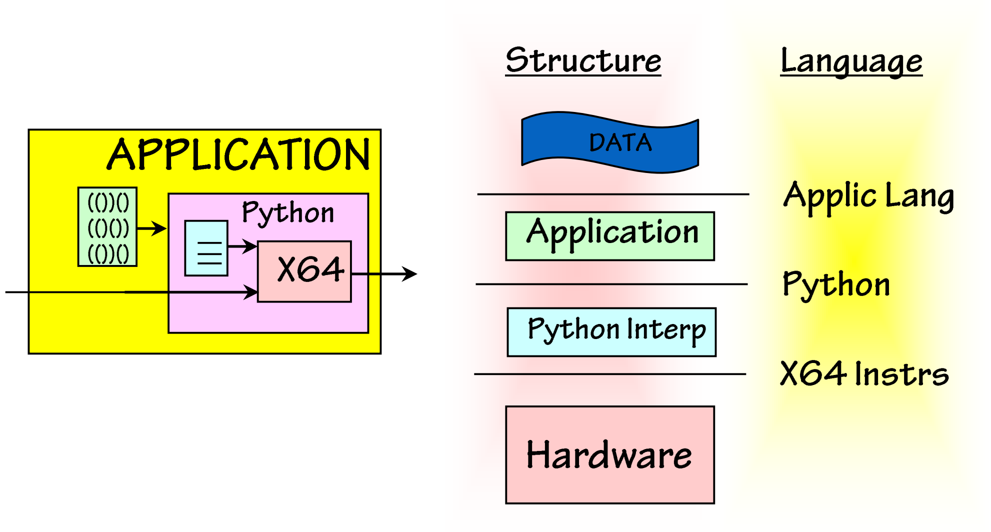
Interpretation and Compliation

Turing's model of *Interpretation:*

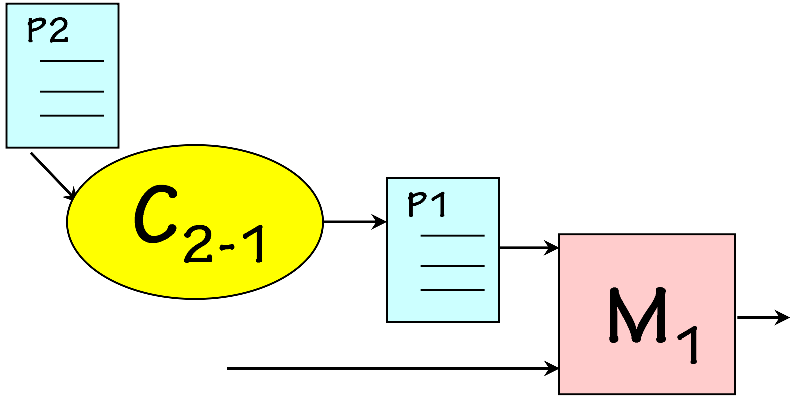


- Start with some hard-to-program *universal* machine, say M_1
- Write a single program for M_1 which mimics the behavior of some easier machine, say M_2
- Result: a "virtual" M_2

"Layers" of interpretation:



Model of *Compilation:*



- Given some hard-to-program machine, say M_1...
- Find some easier-to-program language L_2 (perhaps for a more complicated machine, M_2); write programs in that language
- Build a translator (compiler) that translates programs from M_2's language to M_1's language. May run on M_1, M_2, or some other machine.

Interpretation vs. Compliation

Interpretation & Compilation improve programmability!

Both ...

- allow changes in programming model
- afford programming applications in platform (e.g., processor) independent languages
- are widely used in modern computer systems!

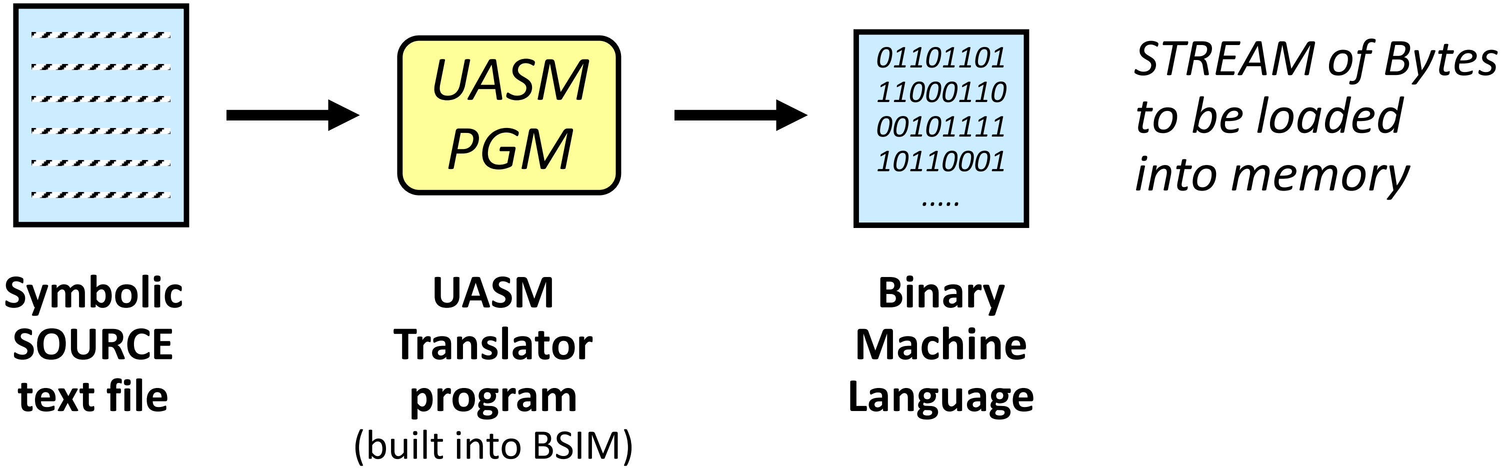
	Interpretation	Compilation
How it treats input "x+2"	computes x+2	generates a program that computes x+2
When it happens	During execution	Before execution
What it complicates/slows	Program Execution	Program Development
Decisions made at	Run Time	Compile Time

Design choice: do it at Compile time or at Run time?

Software Abstraction Strategy

- Initial steps: **compilation tools**
  - Assembler (**UASM**): symbolic representation of machine language
    Hides: bit-level representations, hex locations, binary values
  - Compiler (**C**): symbolic representation of algorithm
    Hides: Machine instructions, registers, machine architecture
- Subsequent steps: **interpretive tools**
  - Operating system
    Hides: Resource (memory, CPU, I/O) limitiations and details
  - Apps (e.g., Browser)
    Hides: Network, location, local parameters

UASM Assembly Language



UASM is

- a program for writing programs ☺
- a symbolic **LANGUAGE** for representing strings of bits
- a **PROGRAM** for translating UASM source to binary (“assembler” = primitive compiler)

See beta.uasm in lab files!

UASM Source Language: Byte values

UASM source (text) file:	Translated byte code:	in hex:
`-3 127 0b1010 0xA9`	`10101001 00001010 01111111 11111101`	`0xA90A7FFD`
`37+0b10-0x10   24-0x1   4*0b110-1`	`00010111 00010111 00010111 00010111`	`0x17171717`
`0xF7&0x1F   33`	`00100001`	`0x00000021`

- Values of successive **bytes** to be loaded into memory
- Interpreted from **left to right** as **least to most significant bytes**
- Values can be decimal, binary (0b), hexadecimal (0x) or **expressions** (+,-,*,/,%,<<,>>,&,|)

UASM Source Language: Symbols

UASM source (text) file:

```
a = 0x1000        | an address
x = 123           | a variable
R0 = 0            | a register
. = 0x1004
   1  0xF3  x  x+4
y: x<<4  1  2  255
. = a
   LONG(y - a)
```

Symbol	value
a	0x1000
x	123
R0	0
y	0x1008

Translated byte code in memory (in hex):

```
0x1000: 00 00 00 08
0x1004: 7F 7B F3 01
0x1008: FE 02 01 B0
```

- **Symbols** (`x = …`) for values, stored in symbol table
- References to **current byte address** (`.`),
- **Labels** (`y:`) symbols that take the value of current memory address

UASM Source Language: Macros

- **Macros** are parameterized symbols:

```
.macro consec(n) n n+1 n+2 n+3
consec(10)                          → 0D 0C 0B 0A
```

Confusing! 32-bit is the word size of the β!

- Macros for writing 16-bit (`WORD`) and 32-bit (`LONG`) **words**:

```
.macro WORD(x) x%256 (x/256)%256
WORD(0x1234)                        → 12 34
WORD(345)                           → 01 59

.macro LONG(x) WORD(x) WORD(x>>16)
LONG(0x123456)                      → 00 12 34 56
```

- **Little/big endian formats**: least/most significant byte is stored at lowest memory address:

```
. = 0x0
  1 2 3 4
```

<u>Little endian:</u> (used in β)

0x3	0x2	0x1	0x0	
04	03	02	01	0x0
…	…	…	…	0x4

<u>Big endian:</u>

	0x0	0x1	0x2	0x3
0x0	04	03	02	01
0x4	…	…	…	…

Assembly of β instructions

-32768 = 1000000000 15= 00000

OPCODE	RC	RA	RB	UNUSED
110000	00000	01111	1000000000000000	

```
| Assemble Beta op instructions
.macro betaop(OP,RA,RB,RC) {
     .align 4
     LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+((RB%32)<<11))
}

| Assemble Beta opc instructions
.macro betaopc(OP,RA,CC,RC) {
     .align 4
     LONG((OP<<26)+((RC%32)<<21)+((RA%32)<<16)+(CC%0x10000))
}

| Assemble Beta branch instructions
.macro betabr(OP,RA,RC,LABEL)     betaopc(OP,RA,((LABEL-(.+4))>>2),RC)
```

".align 4" ensures instructions will begin on word boundary (i.e., address = 0 mod 4)

<u>For Example:</u>

```
     ADDC(R15, -32768, R0) --> betaopc(0x30,15,-32768,0)
```

The β instructions

```
| BETA Instructions:
.macro ADD(RA,RB,RC)          betaop(0x20,RA,RB,RC)
.macro ADDC(RA,C,RC)          betaopc(0x30,RA,C,RC)
…
.macro LD(RA,CC,RC)           betaopc(0x18,RA,CC,RC)
.macro LD(CC,RC)              betaopc(0x18,R31,CC,RC)
.macro ST(RC,CC,RA)           betaopc(0x19,RA,CC,RC)
.macro ST(RC,CC)              betaopc(0x19,R31,CC,RC)
…
.macro BEQ(RA,LABEL,RC)       betabr(0x1D,RA,RC,LABEL)
.macro BEQ(RA,LABEL)          betabr(0x1D,RA,r31,LABEL)
…
```

More convenience macros (`BR, JMP, BF, BT, LDR, MOVE, PUSH, POP, CALL, …`) … see beta.uasm!

Example assembly

```
ADDC(R3,1234,R17)
```

expand `ADDC` macro with `RA=R3, C=1234, RC=R17`

```
betaopc(0x30,R3,1234,R17)
```

expand `betaopc` macro with `OP=0x30, RA=3, CC=1234, RC=17`

```
.align 4
LONG((0x30<<26)+((17%32)<<21)+((3%32)<<16)+(1234 % 0x10000))
```

expand `LONG` macro with `X=0xC22304D2`

```
WORD(0xC22304D2)    WORD(0xC22304D2 >> 16)
```

expand first `WORD` macro with `X=0xC22304D2`

```
0xC22304D2%256    (0xC22304D2/256)%256    WORD(0xC223)
```

evaluate expressions, expand second `WORD` macro with `X=0xC223`

```
0xD2    0x04    0xC223%256    (0xC223/256)%256
```

evaluate expressions

```
0xD2    0x04    0x23    0xC2
```

Example assembler code

Add two numbers $x = 35$ and $y = 99$:

Assembly

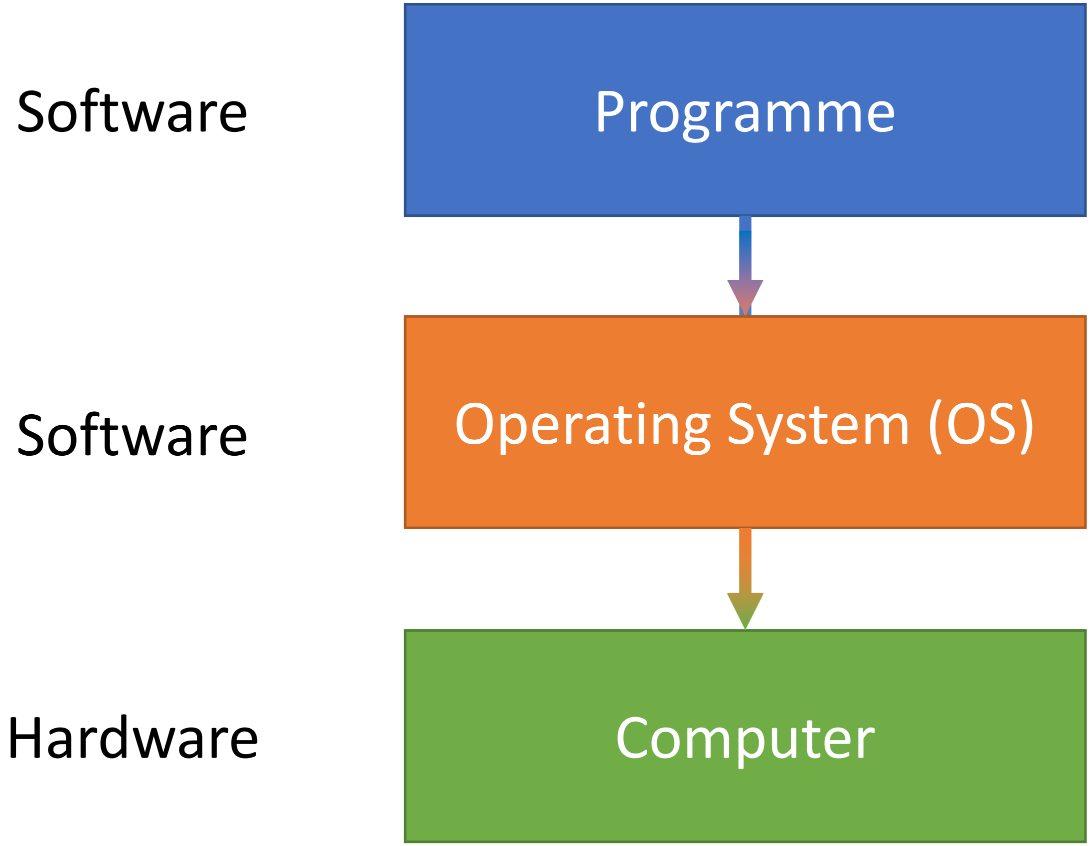
Assembler code: → Instructions in memory:

```
    LD(x,R1)          60 3F 00 14   0x0000   011000 00001 11111 0000000000010100
    LD(y,R2)          60 5F 00 18   0x0004   011000 00010 11111 0000000000011000
    ADD(R1,R2,R0)  code  80 01 10 00   0x0008   100000 00000 00001 00010 00000000000
    ST(R0,Z)          64 1F 00 1C   0x000C   011001 00000 11111 0000000000011100
    HALT()            00 00 00 00   0x0010   00000000 00000000 00000000 00000000

x:  LONG(35)          00 00 00 23   0x0014   00000000 00000000 00000000 00010011
y:  LONG(99)    data  00 00 00 63   0x0018   00000000 00000000 00000000 01100011
z:  LONG(0)           00 00 00 00   0x001C   00000000 00000000 00000000 00000000
```

→ Bsim demo

Programmes and Operating Systems



- Assembly code
- Compiled local: C, C++, Fortran, …
- Interpreted: Java, Python, …

- Loaded at reset and running perpetually
- Manages programmes
- Virtual machines for interpreting code

- Writing an OS in assembly is tedious, since instructions are chip/machine-dependent

→ We need machine-independent, **high-level programming languages**

- **C language**, developed at Bell Labs in 1972 for programming the `UNIX` OS

→ (Machine-dependent) **Compiler** that directly translates `C` code into byte code

C language overview: an example

```
int f(int a, int b) {
    int c=1;
    return a+b+c;
}
int g(int a, int b) {
    return a-b;
}
int main(int argc, char* argv[]) {
    int a,b;
    printf("a=");
    scanf("%d", &a);
    printf("b=");
    scanf("%d", &b);
    if (a>b)
        printf("%d\n", g(a,b));   // returns a-b
    else
        printf("%d\n", f(a,b));   /* returns a+b+1 */
    return 0;
}
```

procedure identifier

return type

procedures (functions)

programme entry point

output to console/screen

input from console/keyboard

conditional (control statement)

(local) variable identifier

literal

expression

operator

parameter

data type

statement

reference to address

comments

Summary

- Software abstraction: Interpretation and Compliation
- Assembler (UASM): symbolic representation of machine language
  - Values of successive bytes to be loaded into memory
  - Symbols, labels, macros
  - → Assembly of 32-bit $\boldsymbol{\beta}$ instructions
- High-level languages and compilation