10a. Assembly Language

6.004x Computation Structures
Part 2 – Computer Architecture

Beta ISA Summary

- Storage:
  - Processor: 32 registers (r31 hardwired to 0) and PC
  - Main memory: Up to 4 GB, 32-bit words, 32-bit byte addresses, 4-byte-aligned accesses
- Instruction formats:

OPCODE	r_c	r_a	r_b	*unused*

OPCODE	r_c	r_a	16-bit signed constant

← 32 bits →

- Instruction classes:
  - ALU: Two input registers, or register and constant
  - Loads and stores: access memory
  - Branches, Jumps: change program counter

Programming Languages

32-bit (4-byte) ADD instruction:

opcode	rc	ra	rb	(unused)
1 0 0 0 0 0	0 0 1 0 0	0 0 0 1 0	0 0 0 1 1	0 0 0 0 0 0 0 0 0 0 0

Means, to the BETA, `Reg[4] ← Reg[2] + Reg[3]`

We'd rather write in *assembly language*:

```
ADD(R2, R3, R4)
```

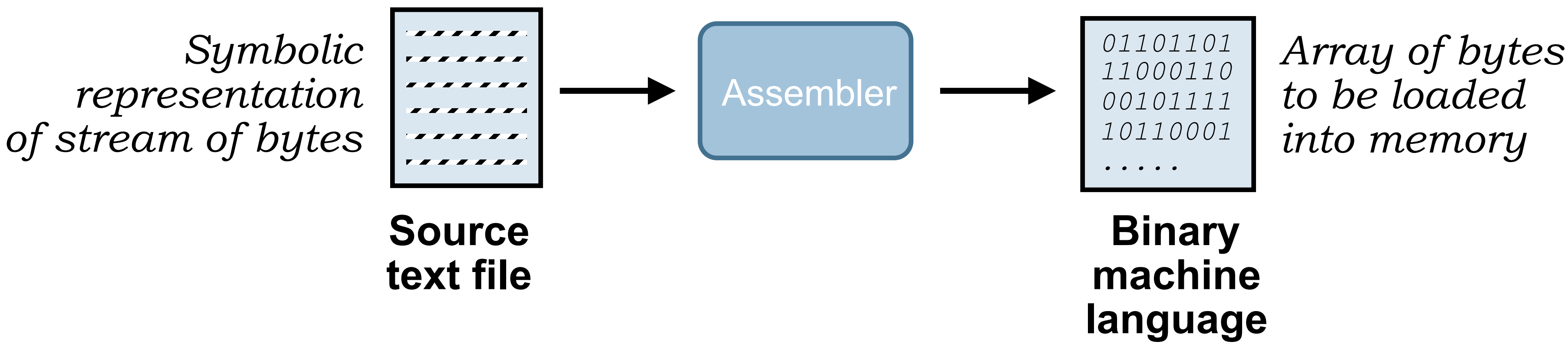
Today

or better yet a *high-level language*:

```
a = b + c;
```

Coming up

Assembly Language



- Abstracts bit-level representation of instructions and addresses
- We'll learn UASM ("microassembler"), built into BSim
- Main elements:
  - Values
  - Symbols
  - Labels (symbols for addresses)
  - Macros

Example UASM Source File

```
      N = 12                  // loop index initial value
      ADDC(r31, N, r1)        // r1 = loop index
      ADDC(r31, 1, r0)        // r0 = accumulated product
loop: MUL(r0, r1, r0)         // r0 = r0 * r1
      SUBC(r1, 1, r1)         /* r1 = r1 - 1 */
      BNE(r1, loop, r31)      // if r1 != 0, NextPC=loop
```

- Comments after //, ignored by assembler (also /*…*/)
- Symbols are symbolic representations of a constant value (they are NOT variables!)
- Labels are symbols for addresses
- Macros expand into sequences of bytes
  - Most frequently, macros are instructions
  - We can use them for other purposes

How Does It Get Assembled?

Text input

```
        N = 12
        ADDC(r31, N, r1)
        ADDC(r31, 1, r0)
loop:   MUL(r0, r1, r0)
        SUBC(r1, 1, r1)
        BNE(r1, loop, r31)
```

Binary output

```
110000 00001 11111 00000000 00001100 [0x00]
110000 00000 11111 00000000 00000001 [0x04]
100010 00000 00000 00001 00000000000 [0x08]
...
```

- Load predefined symbols into a symbol table
- Read input line by line
  - Add symbols to symbol table as they are defined
  - Expand macros, translating symbols to values first

Symbol table

Symbol	Value
r0	0
r1	1
...	
r31	31
N	12
loop	8

Registers are Predefined Symbols

- r0 = 0, …, r31 = 31
- Treated like normal symbols:

```
ADDC(r31, N, r1)
```

↓ *Substitute symbols with their values*

```
ADDC(31, 12, 1)
```

↓ *Expand macro*

```
110000 00001 11111 00000000 00001100
```

- No "type checking" if you use the wrong opcode…

```
ADDC(r31, r12, r1)
```

↓

```
ADDC(31, 12, 1)
```

Reg[1] ← Reg[31] + 12

```
ADD(r31, N, r1)
```

↓

```
ADD(31, 12, 1)
```

Reg[1] ← Reg[31] + Reg[12]

Labels and Offsets

Input file

```
      N = 12
      ADDC(r31, N, r1)
      ADDC(r31, 1, r0)
loop: MUL(r0, r1, r0)
      SUBC(r1, 1, r1)
      BNE(r1, loop, r31)
```

- Label value is the address of a memory location
- BEQ/BNE macros compute offset automatically
- Labels hide addresses!

Output file

```
110000 00001 11111 00000000 00001100 [0x00]
110000 00000 11111 00000000 00000001 [0x04]
100010 00000 00001 00000 00000000000 [0x08]
110001 00001 00001 00000000 00000001 [0x0C]
011101 11111 00001 11111111 11111101 [0x10]
```

```
offset = (label - <addr of BNE/BEQ>)/4 – 1
       = (8 – 16)/4 – 1 = -3
```

Symbol table

Symbol	Value
r0	0
r1	1
...	
r31	31
N	12
loop	8

Mighty Macroinstructions

Macros are parameterized abbreviations, or shorthand

```
// Macro to generate 4 consecutive bytes:
.macro consec(n)  n  n+1  n+2  n+3

// Invocation of above macro:
consec(37)
```

Is expanded to

```
 ⇒ 37 37+1 37+2 37+3 ⇒ 37  38  39  40
```

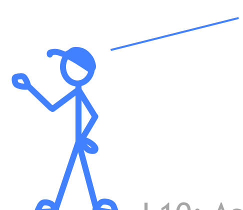
Here are macros for breaking multi-byte data types into byte-sized chunks

```
// Assemble into bytes, little-endian:
.macro WORD(x) x%256 (x/256)%256
.macro LONG(x) WORD(x) WORD(x >> 16)

. = 0x100
 LONG(0xdeadbeef)
```

Has same effect as:

	0xef	0xbe	0xad	0xde
Mem:	0x100	0x101	0x102	0x103



Boy, that's hard to read. Maybe, those big-endian types do have a point.

Assembly of Instructions

-32768 = 1000000000000000

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)

For example:

```
.macro ADDC(RA,C,RC)   betaopc(0x30,RA,C,RC)

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

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=R3, CC=1234, RC=R17

```
.align 4
LONG((0x30<<26)+((R17%32)<<21)+((R3%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
```

UASM Macros for Beta Instructions

(defined in beta.uasm)

```
| BETA Instructions:
.macro ADD(RA,RB,RC)    betaop(0x20,RA,RB,RC)
.macro ADDC(RA,C,RC)    betaopc(0x30,RA,C,RC)
.macro AND(RA,RB,RC)    betaop(0x28,RA,RB,RC)
.macro ANDC(RA,C,RC)    betaopc(0x38,RA,C,RC)
.macro MUL(RA,RB,RC)    betaop(0x22,RA,RB,RC)
.macro MULC(RA,C,RC)    betaopc(0x32,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(0x1C,RA,RC,LABEL)
.macro BEQ(RA,LABEL)    betabr(0x1C,RA,r31,LABEL)
.macro BNE(RA,LABEL,RC) betabr(0x1D,RA,RC,LABEL)
.macro BNE(RA,LABEL)    betabr(0x1D,RA,r31,LABEL)
```

Convenience macros so we don't have to specify R31…

Pseudoinstructions

- Convenience macros that expand to one or more real instructions
- Extend set of operations without adding instructions to the ISA

```
// Convenience macros so we don't have to use R31
.macro LD(CC,RC)        LD(R31,CC,RC)
.macro ST(RA,CC)        ST(RA,CC,R31)
.macro BEQ(RA,LABEL)    BEQ(RA,LABEL,R31)
.macro BNE(RA,LABEL)    BNE(RA,LABEL,R31)

.macro MOVE(RA,RC)            ADD(RA,R31,RC)     // Reg[RC] <- Reg[RA]
.macro CMOVE(CC,RC)           ADDC(R31,C,RC)     // Reg[RC] <- C
.macro COM(RA,RC)             XORC(RA,-1,RC)     // Reg[RC] <- ~Reg[RA]
.macro NEG(RB,RC)             SUB(R31,RB,RC)     // Reg[RC] <- -Reg[RB]
.macro NOP()                  ADD(R31,R31,R31)   // do nothing

.macro BR(LABEL)              BEQ(R31,LABEL)     // always branch
.macro BR(LABEL,RC)           BEQ(R31,LABEL,RC)  // always branch
.macro CALL(LABEL)            BEQ(R31,LABEL,LP)  // call subroutine
.macro BF(RA,LABEL,RC)        BEQ(RA,LABEL,RC)   // 0 is false
.macro BF(RA,LABEL)           BEQ(RA,LABEL)
.macro BT(RA,LABEL,RC)        BNE(RA,LABEL,RC)   // 1 is true
.macro BT(RA,LABEL)           BNE(RA,LABEL)

// Multi-instruction sequences
.macro PUSH(RA)               ADDC(SP,4,SP)  ST(RA,-4,SP)
.macro POP(RA)                LD(SP,-4,RA)   ADDC(SP,-4,SP)
```

Factorial with Pseudoinstructions

Before

```
      N = 12
      ADDC(r31, N, r1)
      ADDC(r31, 1, r0)
loop: MUL(r0, r1, r0)
      SUBC(r1, 1, r1)
      BNE(r1, loop, r31)
```

After

```
      N = 12
      CMOVE(N, r1)
      CMOVE(1, r0)
loop: MUL(r0, r1, r0)
      SUBC(r1, 1, r1)
      BNE(r1, loop)
```

Raw Data

- LONG assembles a 32-bit value
  - Variables
  - Constants > 16 bits

```
N:     LONG(12)
factN: LONG(0xdeadbeef)
       ...
Start:
       LD(N, r1)
       CMOVE(1, r0)
loop:  MUL(r0, r1, r0)
       SUBC(r1, 1, r1)
       BT(r1, loop)
       ST(r0, factN)
```

Symbol table

Symbol	Value
...	
N	0
factN	4

```
LD(r31, N, r1)
   ⇩
LD(31, 0, 1)

Reg[1] ← Mem[Reg[31] + 0]
       ← Mem[0]
       ← 12
```

UASM Expressions and Layout

- Values can be written as expressions
  - Assembler evaluates expressions, they are *not* translated to instructions to compute the value!

```
A = 7 + 3 * 0x0cc41
B = A - 3
```

- The "." (period) symbol means the next byte address to be filled
  - Can read or write to it
  - Useful to control data layout or leave empty space (e.g., for arrays)

```
. = 0x100           // Assemble into 0x100
LONG(0xdeadbeef)
k = .               // Symbol "k" has value 0x104
LONG(0x00dec0de)
. = .+16            // Skip 16 bytes
LONG(0xc0ffeeee)
```

Summary: Assembly Language

- Low-level language, symbolic representation of sequence of bytes. Abstracts:
  - Bit-level representation of instructions
  - Addresses
- Elements: Values, symbols, labels, macros
- Values can be constants or expressions
- Symbols are symbolic representations of values
- Labels are symbols for addresses
- Macros are expanded to byte sequences:
  - Instructions
  - Pseudoinstructions (translate to 1+ real instructions)
  - Raw data
- Can control where to assemble with "." symbol