<html><head></head><body><pre style="word-wrap: break-word; white-space: pre-wrap;">* Representation of data
** Memory is a big array
   No such think as two- (or multi-) dimentional data
** Representing binary integers
   Let's go back to second grade: Place value notation
*** Unsigned
*** Two's Compliment
    Invert bits and add 1 to negate
    One more negative than positive value
    Carries are discarded
*** One's Compliment
    Has two zeros (positive and negative
    Carries must be brought around to ones place
** Hexadecimal
* LEGv8 Assembly Language
** According to the textbook, "LEG" is a pun on "ARM", as well as a backronym
     meaning "Lessen Extrinsic Garrulity"
** ARMv8 is a smallish instruction set, but it still has over 1000
     instructions (most of these are versions of each other). LEGv8 is a
     complete (in the technical sense), pedagogical subset of ARMv8
** There are many LEGv8 emulators out there. We'll be using one of them for
     our first programming assignment, writing some small LEGv8 assembly
     programs.
** Instruction format
   R (register) instructions: Instruction dest reg, first reg, second reg
   I (immediate) instructions: Instruction dest reg, source reg, #immediate
   D (data) instructions: Instruction register, [address register, #imm offset]
** Registers have names X0-X31
   Example: Compile the line of C code: y = x + a[10] + 3
     Where y is in X21, x is in X20, and the address of a is in X22:
   Load to a register with LDUR
   Add two registers with ADD
   Add a register and an immediate with ADDI
     LDUR X21, [X22, #80]
     ADDI X21, X21, #3
     ADD X21, X21, X20
** Jumping
   B (branch) jumps unconditionally, takes a label
   CBZ (compare and branch if zero), takes a register and a label
   CBNZ (compare and branch if not zero, takes a register and a label
   Labels are simply "name:" in the program

   if (x == y)</pre></body></html>

```
    x++;
  else
    y++;

  x -&gt; X21
  y -&gt; X22


  while (str[i])
    i++;

  str -&gt; X21
  i -&gt; X22
  Use LSL (logical shift left) to multiply by 8

** Making a procedure call
  In order to call a procedure, the program must:
    1) Place the parameters - In ARM, the parameters go in registers
X0-X7
    2) Transfer control - That is, jump to the procedure
    3) Allocate storage - For saved registers, "automatic variables"
    4) Perform calculation
    5) Place the result - Return value goes into parameter registers
(X0-X7)
    6) Return control - Restore saved values and jump back to the
caller

  Registers and instructions to facilitate this:
    registers:
      SP (stack pointer, X28)
      FP (frame pointer, X29)
      LR (return address, X30)
    instructions:
      BL address  - Branch and link (jumps to address and stores PC in
LR)
                    PC is "program counter", special register that
stores
                    the address of the current instruction
      BR register - Branch to register (Jumps to LR + 4 (instructions
are
                    32 bits); If no +4, the BL instruction would
execute
                    again, forever.)

  Registers X0-X7 are for parameters and return values (usually only
X0)
    If more parameters are needed, spill to stack
  Registers X9-X15 are temporaries - Callee is not responsible for
saving
    them.  If caller wants them saved, must save them before BL
instruction
```

Registers X19-X27 are saved by callee (if necessary) to stack

To implement a procedure:

```
//label
myprocedure:
//Save saved registers
SUBI SP, SP, #8 * n (number of things need to save)
STUR reg, [SP, #8 * (n - 1)]
...
STUR reg, [SP, #0]
Procedure body
Return value into X0 (maybe more)
//Restore saved registers
LDUR reg, [SP, #0]
...
LDUR reg, [SP, #8 * (n - 1)
ADDI SP, SP, #8 * n
//Return to instruction after procedure call
BR LR
```

To call a procedure:

```
//Save temporaries that you still need to stack (maybe none)
SUBI SP, SP, #8 * n (number of things need to save)
STUR reg, [SP, #8 * (n - 1)]
...
STUR reg, [SP, #0]
BL myprocedure
//Restore saved registers
LDUR reg, [SP, #0]       // Control returns here
...
LDUR reg, [SP, #8 * (n - 1)
ADDI SP, SP, #8 * n
```

Example:

```
long long fib(long long n)
{
  if (n &lt; 2) {
    return n;
  }
  return fib(n - 1) + fib(n - 2);
}

fib:
  //No stack manipulation, because we don't change anything!
  SUBI X9, X0, #2
  CBZ X9, done
  SUBI X9, X0, #1
  CBNZ X9, body
```

```
  done:
    BR LR //Value is already in X0; nothing to copy
  body:

    //stack
    SUBI SP, SP, #24
    STUR X20, [SP, #0]
    STUR X19, [SP, #8]
    STUR LR, [SP, #16]

    //Save paremeter
    ADD X19, X0, XZR
    //Set paremeter for first call, fib(n - 1)
    SUBI X0, X19, #1
    BL fib
    //save return value
    ADD X20, X0, XZR
    //Set paremeter for second call, fib(n - 2)
    SUBI X0, X19, #2
    BL fib
    //Put return in X0
    ADD X0, X0, X20

    //stack
    LDUR X20, [SP, #0]
    LDUR X19, [SP, #8]
    LDUR LR, [SP, #16]
    ADDI SP, SP, #24

    BR LR

** Preventing data races
   LDXR - LoaD eXclusive Register
   STXR - STore eXclusive Retister

   Given:
     LDXR Xn, [Xm, #0]
     //Optionally, other code
     STXR Xp, Xq, [Xm]

   If at any time between the execution of LDXR from Xm and STXR to Xm
the
     data stored in Xm is changed, STRX does not change Xm, and Xq is
set to
     1; otherwise, Xm = Xp and Xq = 0.

   This code does an atomic exchange:
     again:
       LDXR X10, [X20, #0]
       STXR X23, X9, [X20]
       CBNZ X9, again
```

```
      ADD X23, XZR, X10

   A more efficient lock (1 means lock is acquired):
      ADDI X11, XZR, #1
     again:
      LDXR X10, [X20, #0]
      CBNZ X10, again
      STXR X11, X9, [X20]
      CBNZ X9, again

   And to release lock:
      STUR XZR, [X20, #0]
```

* Encoding and decoding instructions

```
  ADD X12, X10, X11
  This is inst Rd, Rn, Rm
```

ADD is an R (register) format instruction: 11 bit opcode, 5 bit Rm, 6 bit shamt, 5 bit Rn, 5 bit Rd

opcode is 10001010000

```
  ADDI X19, X19, #511
  inst  Rd,  Rn, imm
```

ADDI is an I (immediate) instruction: 10 bit opcode, 12 bit immediate, 5 bit Rn, 5 bit Rd

opcode is 1001000100

```
  LDUR X20, [X10, #8]
  inst  Rt,   Rn, offset
```

LDUR is a D (data) instruction: 11 bit opcode, 9 bit offset, 2 bit op, 5 bit Rn, 5 bit Rt

opcode is 11111000010, op is 00

```
  B foo  // Where foo is 4 instructions above
  inst address
```

B is a B (branch) instruction: 6 bit opcode 000101, 26 bit address

opcode 000101

```
  CBZ X23, foo // Where foo is 4 instructions below
  inst Rt, address
```

CBZ is a CB (conditional branch) instruction: 8 bit opcode, 19 bit address

5 bit Rt

opcode is 10110100

MOVK X15, #43690, LSL 32
inst Rd, imm, op

MOVK is an IM (immediate move) instuction: 9 bit opcode, 2 bit op,
16 bit immediate, 5 bit Rd

opcode is 111100101
op is 00, 01, 10, 11 for 0, 16, 32, 48 respectively

```
1111 1000 0000 0100 0000 0010 0010 0001 // STUR X1, [X17, 64]
1110 1010 0000 1011 0000 0001 0100 1001 // ANDS X9, X10, X11
1101 0001 0000 0000 1010 1001 1110 0101 // SUBI X5, X15, 42
1101 0010 1100 1111 1111 1111 1110 1110 // MOVZ X14, 32767, LSL 32
```

* When and why would you write code in assembly?
* ARMv7 was a 32-bit architecture. Why make ARM-v8 64 bits?
* Why are instructions 32 bits?
** Are larger instruction sets more capable? Faster?
</pre></body></html>