Topic 2
Introduction to ISAs for Embedded Systems

The slides used for this lecture were contributed in part

By Tulika Mitra(tulika@comp.nus.edu.sg)

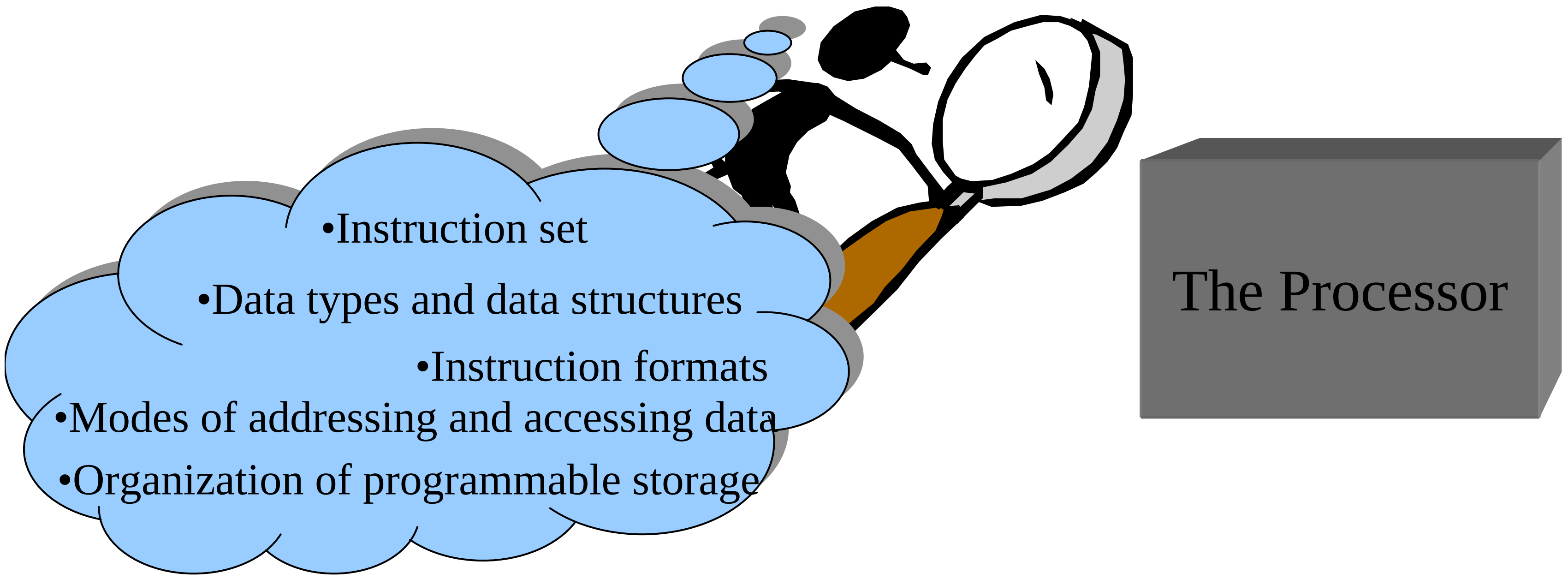
.

What is an ISA?

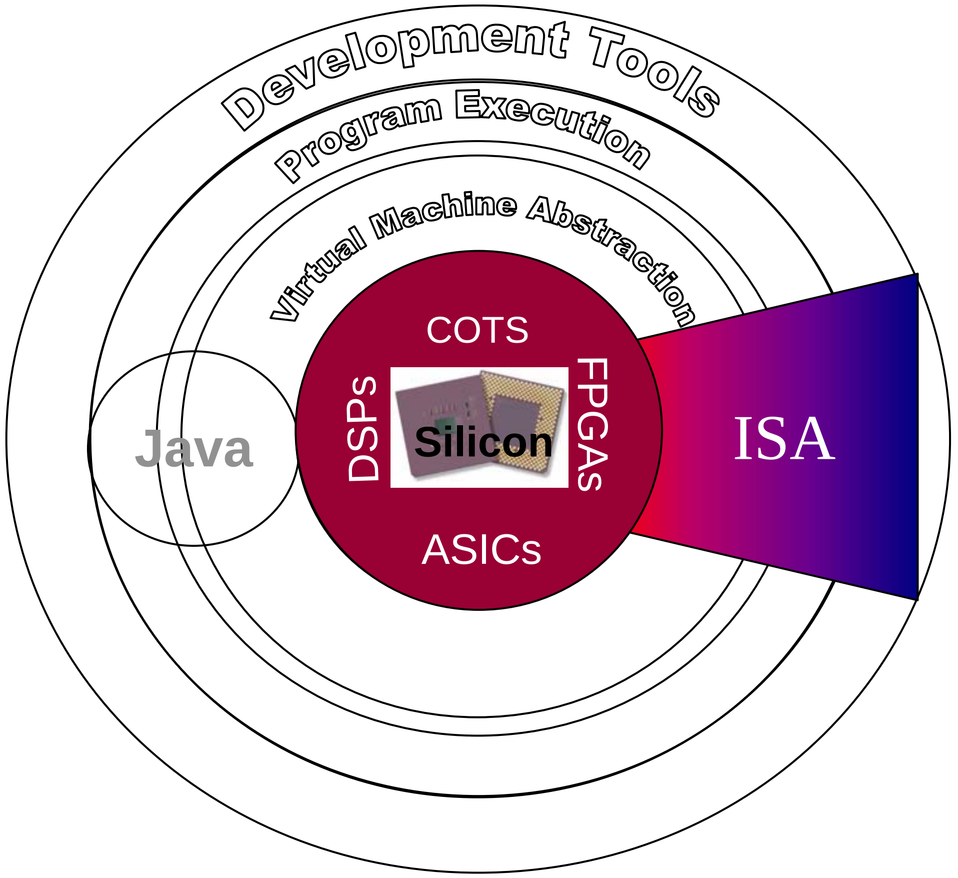
- Definition from Amdahl, Blaaw, and Brooks, 1964

  …the attributes of a [computing] system as seen by the programmer i.e. the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.

  – The ISA gives insight into the makeup of the “Black Box” processor


•Instruction set
•Data types and data structures
•Instruction formats
•Modes of addressing and accessing data
•Organization of programmable storage
The Processor

Where the ISA Fits in the Big Picture



- The ISA is represented by an interface to the processor and its characteristics
  - Visible to various outer rings of software

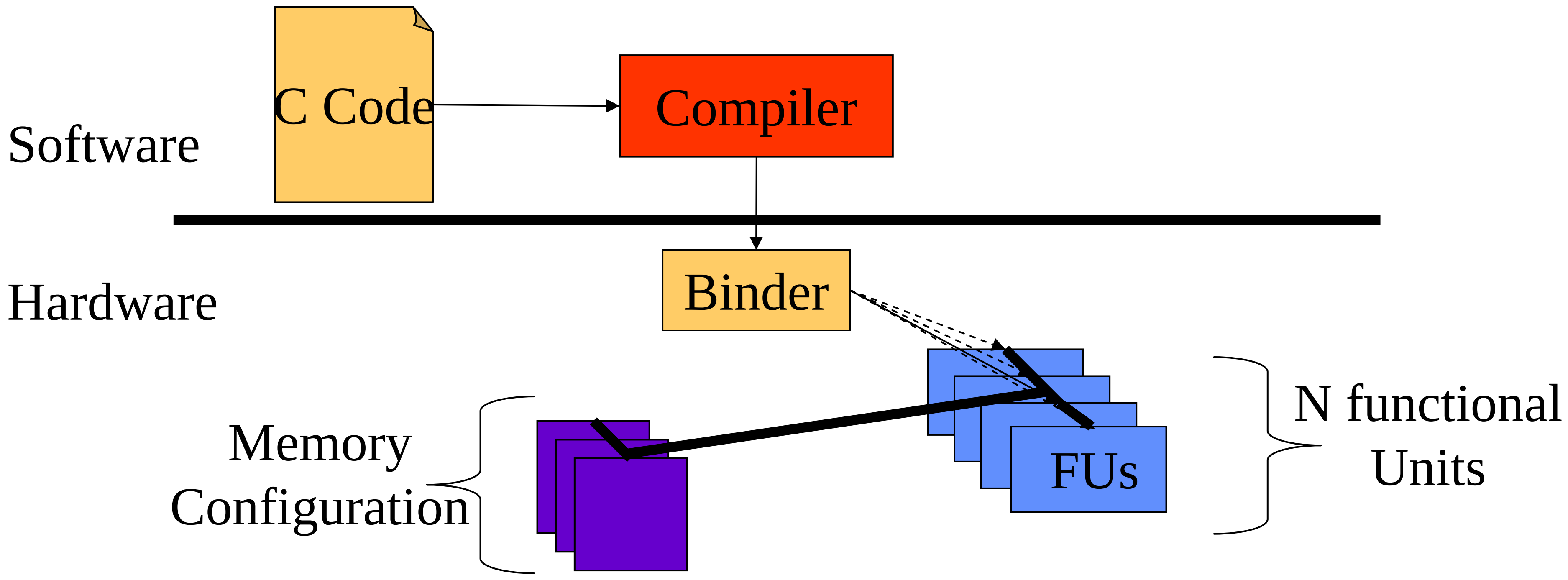
Why are we looking at the ISA?

- Most embedded systems are programmed at the assembly language level.
- Embedded systems compiler designers need to understand the ISA.
  - What are the available data types?
  - How many registers? What types of registers?
  - Can more than one instruction be issued per cycle?
- Embedded systems designers use the ISA to help determine which processor is the best solution.
  - Does the processor provide specialized instructions that are useful?
  - Does the processor provide optimal ways to implement the functions of the application?

ISA View: VLIW vs. Superscalar

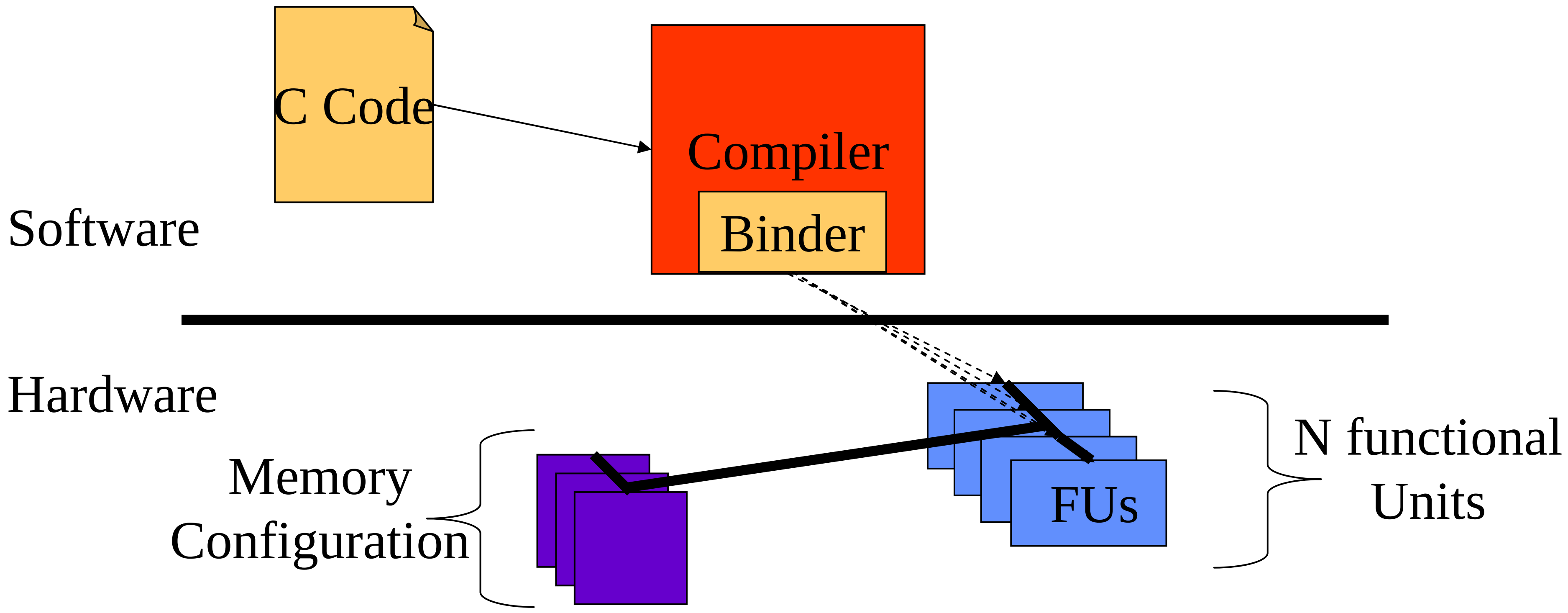
- Another way to contrast VLIW and Superscalar is to consider the views given by the ISAs
  - VLIW gives the assembly language programmer or compiler a view of its functional unit organization
  - Superscalar hides the underlying organization from the programmer and compiler

The Superscalar ISA View



- This ISA only allows a HW binder that maps instructions to FUs after dependence and conflict analysis (dynamic)
- Compiler designer also only sees one FU.
- The HW binder is a very complex piece of HW.

The VLIW ISA View

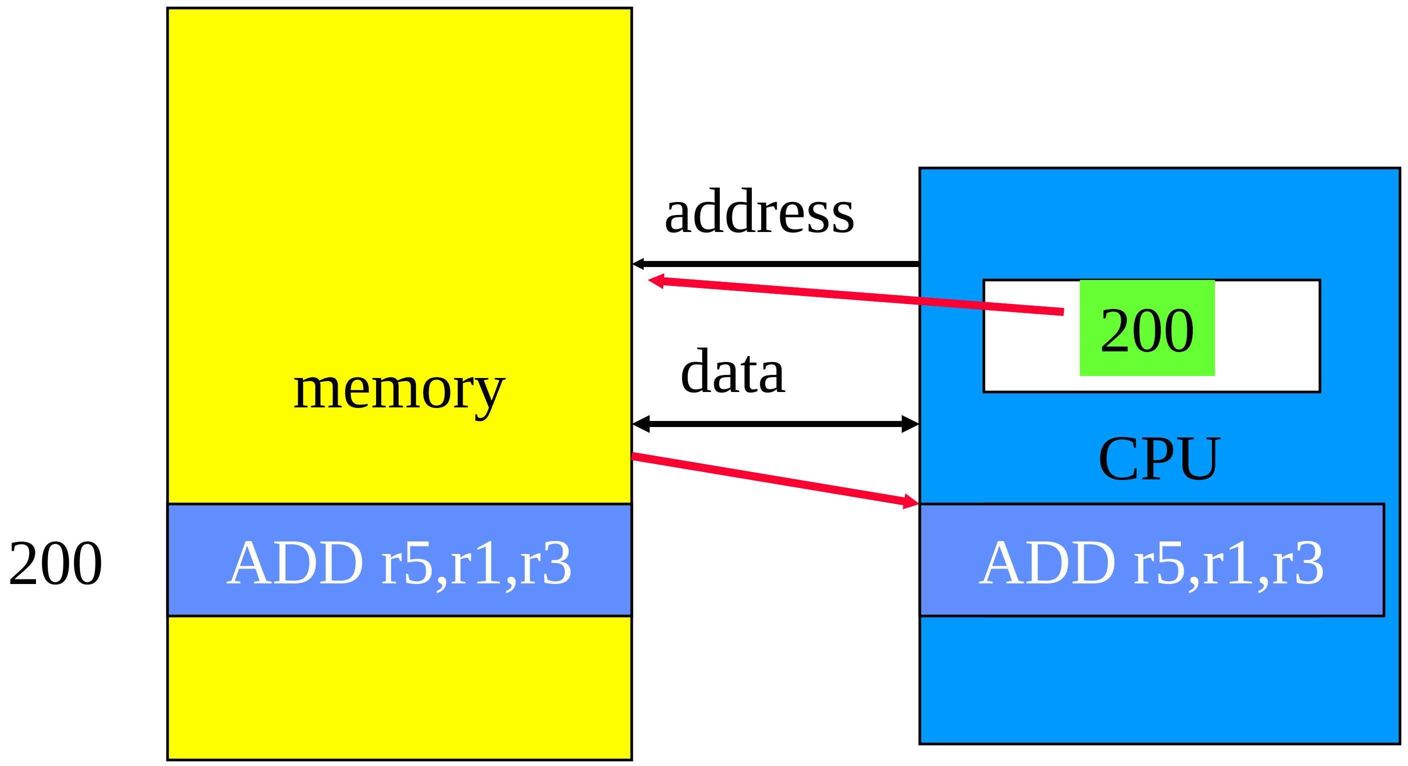


- This ISA enables a SW binder that maps instructions to FUs (Static)
- Compiler designer sees details of FUs
- The SW binder removes HW complexity.

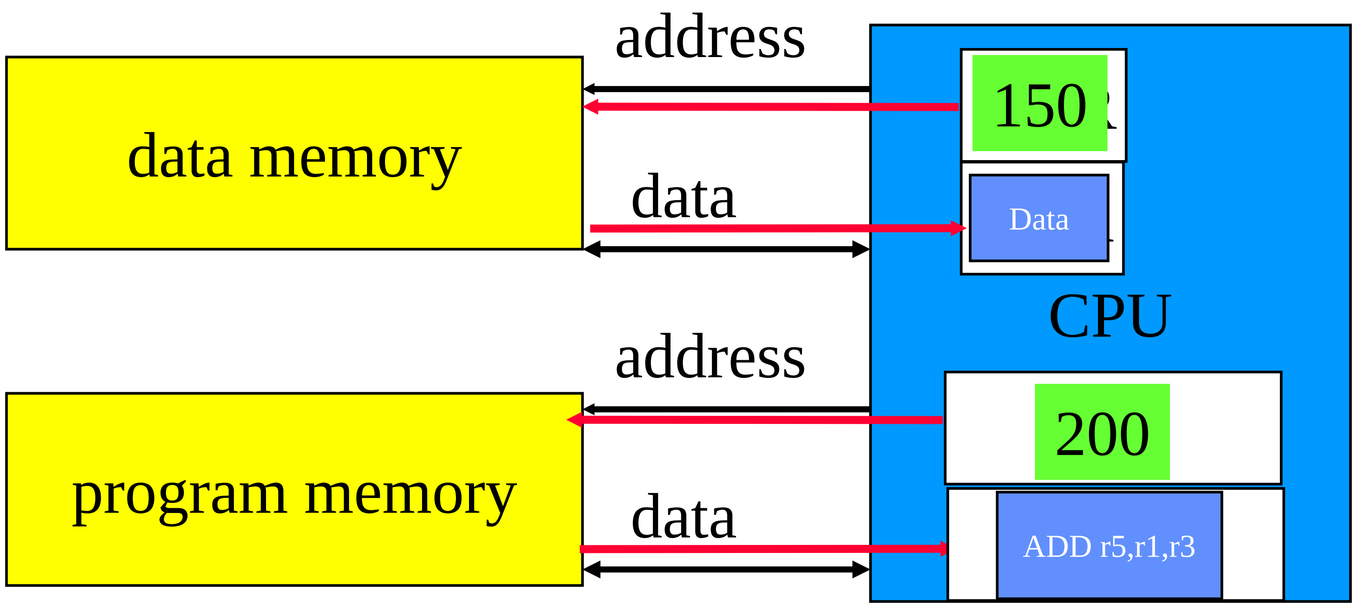
Roadmap thru an ISA

- Micro-architecture
- Assembly language
- Programming model
- Data instructions
- Memory access
- Control flow instructions

Von Neumann Architecture



Harvard architecture



von Neumann vs. Harvard

- Harvard can’t use self-modifying code.
- Harvard allows two simultaneous memory fetches.
- Most DSPs use Harvard architecture for streaming data:
  - greater memory bandwidth;
  - more predictable bandwidth.

ARM vs. SHARC

- ARM7 is von Neumann architecture
  - We will concentrate on ARM7
- ARM9 is Harvard architecture
- SHARC is modified Harvard architecture.
  - On chip memory (> 1Gbit) evenly split between program memory (PM) and data memory (DM)
  - Program memory can be used to store some data.
  - Allows data to be fetched from both memory in parallel

Some Interesting Applications

- ARM
  - Compaq iPAQ
  - Nintendo Gameboy
- SHARC
  - Cellular phone
  - Music synthesis
  - Stereo Receivers

ARM: Assembly Language

```
label1      ADR r4,c
            LDR r0,[r4] ; a comment
            ADR r4,d
            LDR r1,[r4]
            SUB r0,r0,r1 ; comment
```

ARM: Programming Model & Data Types

- Traditional set of registers
  - 15 32-bit General purpose registers
  - Program Counter(PC)
  - Current Program Status Register(CPSR)
    - **Stores condition code bits to record results of comparisons and to control branching**
- The memory system
  - Memory is byte addressable
  - 32-bit addresses
  - Data access can be 8-bit bytes, 16-bit half words, or 32-bit words

ARM: Load/Store Instructions

- LDR, LDRH, LDRB : load (half-word, byte)
- STR, STRH, STRB : store (half-word, byte)
- Addressing modes:
  - register indirect : `LDR r0,[r1]`
  - with second register : `LDR r0,[r1,-r2]`
  - with constant : `LDR r0,[r1,#4]`
  - Base-plus-offset addressing:

    ```
    LDR r0,[r1,#16]
    ```

    - **Loads from location r1+16**
  - Auto-indexing increments base register:

    ```
    LDR r0,[r1,#16]!
    ```

    - **Loads from location r1+16, then r1 = $r1_{old}$ + 16**
  - Post-indexing fetches, then does offset:

    ```
    LDR r0,[r1],#16
    ```

    - **Loads r0 from r1, then adds 16 to r1.**

Example: C Assignments in ARM

- C:

```
x = a + b;
```

- ARM:

```
ADR r4,a             ; get address for a
LDR r0,[r4]          ; get value of a
ADR r4,b             ; get address for b, reusing r4
LDR r1,[r4]          ; get value of b
ADD r3,r0,r1         ; compute a+b
ADR r4,x             ; get address for x
STR r3,[r4]          ; store value of x
```

ARM: Flow of Control

- The Branch Instruction(B) changes flow of control
  - B #100 (adds 400 to the PC)
  - BEQ (branches on equals from CPSR)
  - BGT (branches on greater than in the CPSR)
- Loops, if stmts, switch and case stmts can be implemented with branching
- All operations can be performed conditionally, testing CPSR, by appending the following:
  - EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE

ARM: If Statement Using Branching

C Code: `if (a < b) x = 5; else x = 8;`

ARM Code:

```
        ADR r4,a ; get address for a
        LDR r0,[r4] ; get value of a
        ADR r4,b ; get address for b
        LDR r1,[r4] ; get value for b
        CMP r0,r1 ; compare a, b
        BGE fblock ; if a >= b, branch to false block
        MOV r0,#5 ; generate value for x
        ADR r4,x ; get address for x
        STR r0,[r4] ; store x
fblock:
        MOV r0,#8 ; generate value for x
    ADR r4,x ; get address for x
    STR r0,[r4] ; store value of x
```

ARM: If stmt Using Conditional Instructions

C Code: `if (a < b) x = 5; else x = 8;`

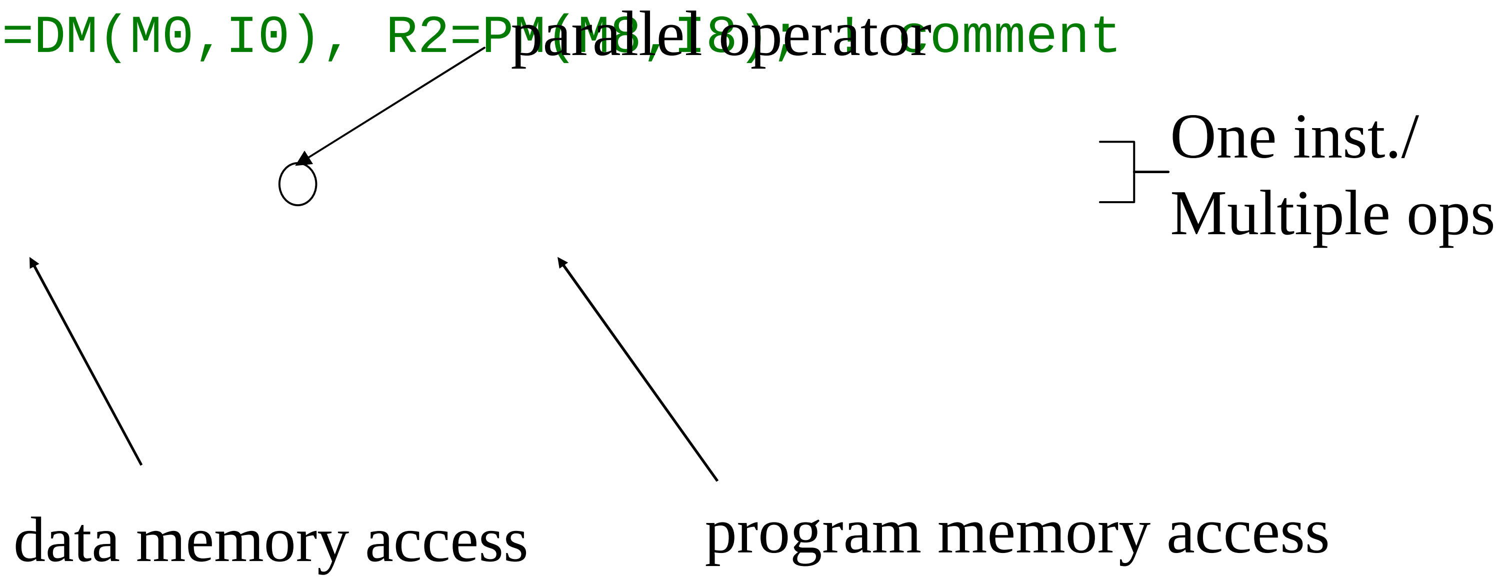
ARM Code:

```
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
ADR r4,b ; get address for b
LDR r1,[r4] ; get value for b
CMP r0,r1 ; compare a, b
MOVLT r0,#5 ; generate value for x if r0 < r1
ADRLT r4,x ; get address for x if r0 < r1
STRLT r0,[r4] ; store x if r0 < r1
MOVGE r0,#8 ; generate value for x if r0 >= r1
ADRGE r4,x ; get address for x if r0 >= r1
STRGE r0,[r4] ; store value of x if r0 >= r1
```

SHARC: Assembly Language

- Algebraic notation terminated by semicolon:

```
R1=DM(M0,I0), R2=PM(M8,I8); ! comment
```



SHARC: Multifunction Computations

Can issue some computations in parallel:

- dual add-subtract;
- fixed-point multiply/accumulate and add,subtract,average
- floating-point multiply and ALU operation
- multiplication and dual add/subtract

SHARC: Programming Model

- Registers for integer and floating point
  - 16 40-bit data registers for integer operations R0-R15
  - R0-R15 are aliased as F0-F15 for floating point operations
- A set of control registers
  - Status registers
  - Loop registers
    - **Supports special loop instructions**
  - Data address generator registers
    - **Assist in handling the loading and storing of data**
  - Interrupt registers
- Functional Units
  - ALU, multiplier & shifter
- Memory is organized as 32-bit words

SHARC: Data Types

- 32-bit IEEE single-precision floating-point
- 40-bit IEEE extended-precision floating-point
- 32-bit integers

SHARC: Load/Store Instructions

- Load/store architecture
  - no memory-direct operations; all data must be loaded into registers
- Two data address generators(DAG):
  - PM: program memory
  - DM: data memory
- Must set up DAG registers to control loads/stores
  - DAG registers automatically update to give quick access to arrays (example later)

SHARC: Addressing Modes

- Immediate value:
  - `R0 = DM(0x20000000);`
- Direct load
  - `R0 = DM(_a); ! Loads contents of _a`
- Direct store
  - `DM(_a)= R0; ! Stores R0 at _a`
- Base-Plus-Offset
  - R0 = DM(M1, I0); ! Loads from location I0 + M1
    - **M and I are registers from the DAG register file**

SHARC: Flexible Loading/Storing

- Compiler allows programmer to control where data is placed in memory
  - Either data memory or program memory

```
float dm a[N];  // put a in data memory
float pm b[N];  // put b in program memory
```

- Two data loads in one cycle:

```
F0 = DM(M0,I0), F1 = PM(M8,I9);
```

Example: C assignments in SHARC

- C:

```
x = a + b;
```

- SHARC:

```
R1 = DM(_a)  ! Load a
R2 = DM(_b); ! Load b
R0 = R1 + R2;
DM(_x) = R0; ! Store result in x
```

SHARC: Example, cont’d.

- Shorter version using pointers:

```
R2=DM(I1,M5), R1=PM(I8,M13);
R0 = R2+R1;
DM(I0,M5)=R0; ! Store in x
```

- The variable b is in DM and a is in PM

SHARC: Flow of Control

- The jump is the basic mechanism
  - Jumps can be based on direct addressing
  - Jumps can be Indirect addressed
    - **Uses DAG2(PM) registers**
  - Jumps can be PC-relative
- All Instructions may be executed conditionally
- Conditions come from:
  - arithmetic status (ASTAT)
  - mode control 1 (MODE1)
  - loop register

SHARC: If stmt implemented using ILP and Conditional Instructions

C Code: `if (a>b) y = c-d; else y = c+d;`

SHARC Code:

```
! Load values
R1=DM(_a);
R2=DM(_b);
R3=DM(_c);
R4=DM(_d);
! Compute both sum and difference
R0=R3 + R4, R12 = R3 – R4;
! Choose which one to save
comp(R1,R2);
if GT R0=R12;
dm(_y) = R0 ! Write to y
```

SHARC: DO UNTIL loops

- DO UNTIL instruction provides efficient looping:

```
                LCNTR=30, DO label UNTIL LCE;
                R0=DM(I0,M0), F2=PM(I8,M8);
Loop length     R1=R0-R15;
    label: F4=F2+F3;
```

Label for Last instruction in loop

Termination Condition (LCE = Loop Counter Expired)

- Arrays are stored at M8 and M0
- The I0 and I8 registers are automatically incremented, while LCNTR is decremented

Summary

- The ISA provides the interface for the embedded systems programmers and compiler designers
  - The movement is towards VLIW style: simpler hardware exploited by aggressive compiler algorithms
- Knowing the ISA helps an embedded systems designer decide which platform to incorporate in their design