Hardware and Instruction Set Architecture

CSE 132

1

Assignment Logistics

- Assignment 5 last demo day is today in OH
  - 9 days of break don't count as late tickets
- Assignment 6 help was posted earlier this week
  - 2 additional late tickets provided for everyone
  - Use for assignment 6 or wherever it benefits you
- Assignment 7 due date pushed 1 week
  - Now due Mar 31
  - Quiz 7B also due Mar 31

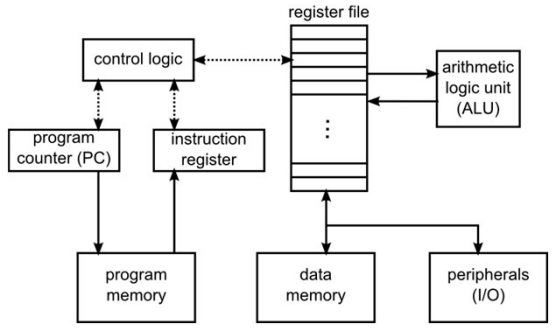
2

Schedule Logistics

- Last 3 modules of semester will focus on assembly language
  - Lecture today, next Wed, and Apr 9 (after exam 2)
  - Three assignments, first due Apr 7
  - There will be quizzes on this material
- Lecture next Wed includes review for exam 2
  - Material in modules 4 to 7
  - April 2, in class
- Last lecture (Apr 16) will be review for exam 3
  - Material will *not* be cumulative, just modules 8 to 10
  - April 23, in class

3

Simple Computer System



For Arduino, all of this is in a single AVR chip.
Chips of this type are called "microcontrollers"

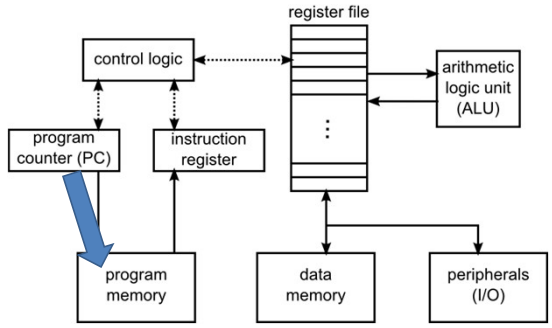
4

Fetch-Decode-Execute Cycle

- Fetch: grab (fetch) the instruction to be executed. It's address is in the instruction pointer (IP) or program counter (PC)
- Decode: figure out what instruction it is and what is to be done (e.g., this is an ADD inst. that needs two values from the register file)
- Execute: do the real work and store the result somewhere (as told by the instruction)

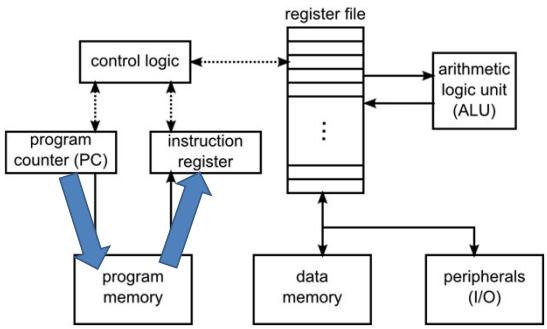
5

Fetch



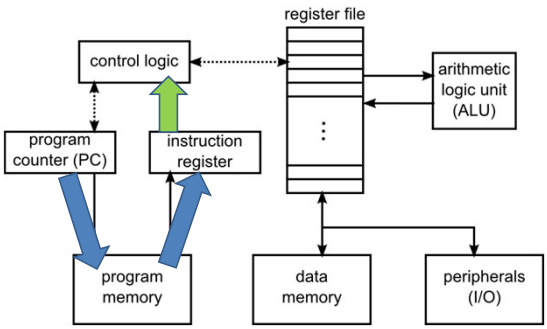
6

Fetch



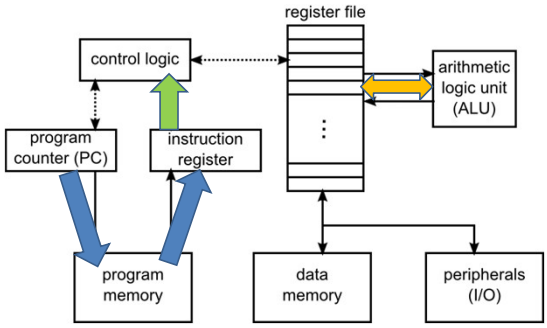
7

Decode



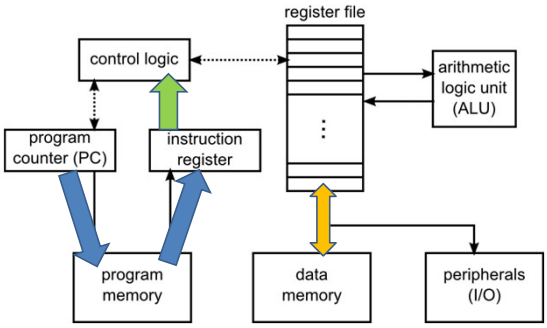
8

Execute



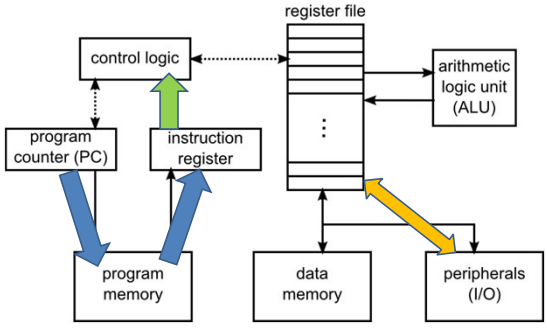
9

Execute (2)



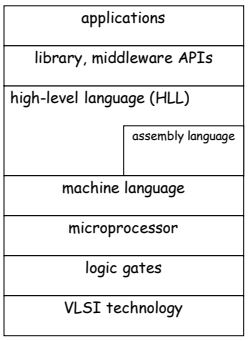
10

Execute (3)



11

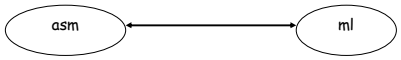
Abstraction Levels



12

HLLs, assembly, vs. machine language

- machine language = binary (i.e., computer readable) image of program code
- assembly language = human readable (and writeable) syntax directly representing machine language

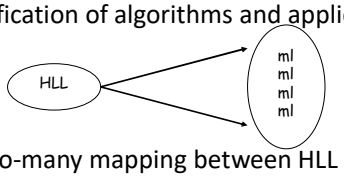


- one-to-one mapping between asm and machine language

13

HLLs, assembly, vs. machine language

- high-level language = designed for human specification of algorithms and applications



- one-to-many mapping between HLL and machine language
- Assembly/machine language is very much architecture dependent
- HLLs are (largely) architecture independent

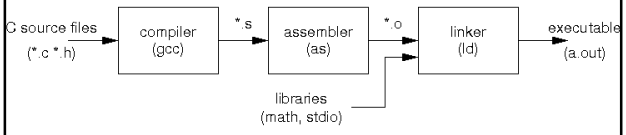
14

Why use assembly language?

- Direct control over the hardware
  - "I want the machine to execute these exact instructions."
- Historical reasons
  - execution efficiency (speed, code size, etc.)
  - lack of suitable HLL compiler (embedded)
- Today
  - Limited need as an authoring language
  - Very useful for investigation – understanding!

15

gcc workflow



16

Instruction Set Architecture (ISA)

- Programmer's view of the processor. It includes the following components:
  - Instruction set: the collection of instructions that are supported by the processor.
  - Register file: the programmer-visible storage within the processor.
  - Memory: the logical organization of the memory (again, programmer's view)
  - Operating modes: some processors have subsets of the instructions that are privileged based on being in a given "mode." (The Arduino AVR processor doesn't have this, but the x86 processor inside a PC does.)

17

AVR Instruction Set

- Arithmetic operations: (add, sub, mul, etc.)
- Boolean operations: (and, or, etc.)
- Shift operations: (left shift, right shift)
- Comparison operations: (<, ≤, >, ≥, =, ≠)
- Memory operations: (load, store)
  - Data movement operations are the only ops that reference memory, all others are to/from registers

18

AVR Instruction Set

- Control flow operations:
  - Unconditional branch: (jmp)
  - Conditional branch: (breq, brne, etc.)
  - Procedure call/return: (call, ret)
- Peripheral access: (in, out)
- System operations: (nop, sleep, etc.)

19

AVR Register File

- 32 general-purpose registers in the AVR ISA:
  - Each 8 bits wide, named R0 to R31
  - Sometimes paired for 16-bit data – e.g., (R5:R4) has least significant bits in R4 and msbits in R5
  - Last 3 register pairs used for addressing – they are named X (R27:R26), Y (R29:R28), and Z (R31:R30)
- 3 special-purpose registers
  - PC – program counter (16 bits wide)
  - SREG – Status register (8 bits wide)
  - SP – stack pointer (16 bits wide), for system stack

20

Status Register

- SREG is status register ⇒ status bits retaining results of previous operations:
  - C – carry – result of unsigned add is too large
  - Z – zero – result of previous operation is 0
  - N – negative – result of operation is negative
  - V – overflow – result of signed op is out of range
  - S – sign – true sign = N xor V
  - H – half carry – used for BCD arithmetic
  - T – bit copy – used by bit load and store inst.
  - I – interrupt – interrupts are enabled

21

General Form

label: opcode operands comment

- Label is optional
- Opcode is the specific instruction (e.g., add)
- Operands specify data for operation
  - AVR is 2-operand machine, 1st operand is dest.
- Comments use different notations
  - Many assemblers (incl. AVR) ; comment
  - Or some other notation, e.g., # comment

22

Pseudo-operations

Pseudo-ops are commands to assembler

.text	means "text section", or instructions are next
.data	means "data section"
.byte	reserves data storage

```
var:  .byte  10
```

reserves one byte, initializes it to 10, and makes var a label that is address of byte

23

Example

```
byte rshift2(byte x) {
      return(x >> 2);
}
```

```
      .text             ;code segment follows
      .global rshift2   ;tell linker about rshift2
rshift2:
      lsr r24           ;do actual work
      lsr r24           ;result is in r24
      ldi r25, 0        ;return value in r25:r24
      ret               ;return
```

24

Addressing Modes

- Register addressing – operand is in register

```
add   Rd, Rr        Rd ← Rd + Rr
 ↑    ↑   ↑
 |    |   └ source operand
 |    └ destination operand
 └ operation (op code)
```

$0 \le d \le 31, 0 \le r \le 31$

25

- Immediate addressing – operand is explicitly present in code

```
subi  Rd, 10        Rd ← Rd - 10
       ↑
       └ "immediate" operand
```

- Constant values use C notation:
  - Default base is 10
  - Hex uses 0x notation (10_{16} is written 0x10)
  - Negative constants are allowed, e.g., -12

26

Cautions

- Assembly has no understanding of data type
  - Programmer must handle multi-byte data
  - No conversions, Load (ld) just copies bits in memory to same bits in register
- Addresses are 16 bits
  - Requires two registers (r31:r30) and two loads
    - lo8(x) gives low byte of x, hi8(x) gives high byte of x
    - Use Load Immediate (ldi), because x is the address

27

Assembly and C

Each can call the other, but assembly routine must follow rules set by C compiler

- r0 is temporary, alter with impunity
- r1 is zero, if changed in assembly, change back
- r2 to r17 and r28 to r29 are callee save
  - Called routine must save if it wishes to use register
- r18 to r27 and r30 to r31 are caller save
  - Calling routine must save register if value is to be preserved across the call

29

Register Usage Conventions in AVR C

Register	Description	Assembly code called from C (callee)	Assembly code that calls C (caller)
r0	Temporary	Save and restore if using	Save and restore if using
r1	Always zero	Must clear before returning	Must clear before calling
r2-r17	"callee-save"	Save and restore if using	Can freely use
r28			
r29			
r18-r27	"caller-save"	Can freely use	Save and restore if using
r30			
r31			

30

Parameters and Return Values

- Two-byte return values go in r25:r24
- Parameters go in register pairs
  - First parameter in r25:r24
  - Second param. in r23:r22
  - Third param. in r21:r20
  - Etc.
- One-byte return values and parameters use low byte of two-byte register pairs

31

Multi-byte Data Manipulation

- Use bits in SREG to save intermediate values
- *C* bit (carry) for addition, e.g,
  r9:r8 ← r9:r8 + r5:r4

```
add r8, r4     ;adds lsbits and puts carry in C
adc r9, r5     ;uses carry from prev. add
```

32