F28HS2 Hardware-Software Interfaces

Lecture 6: ARM Assembly Language 1

CISC & RISC

- CISC: complex instruction set computer
- original CPUs very simple
- poorly suited to evolving high level languages
- extended with new instructions
  - e.g. function calls, non-linear data structures, array bound checking, string manipulation
- implemented in *microcode*
  - sub-machine code for configuring CPU sub-components
  - 1 machine code == > 1 microcode
- e.g. Intel X86 architecture

CISC & RISC

- CISC introduced integrated combinations of features
  - e.g. multiple address modes, conditional execution
  - required additional circuitry/power
  - multiple memory cycles per instruction
  - over elaborate/engineered
    - many feature combinations not often used in practise
- could lead to loss of performance
- better to use combinations of simple instructions

CISC & RISC

- RISC: reduced instruction set computer
- return to simpler design
- general purpose instructions with small number of common features
  - less circuitry/power
  - execute in single cycle
- code size grew
- performance improved
- e.g. ARM, MIPS

ARM

- Advanced RISC Machines
- UK company
  - originally Acorn
  - made best selling BBC Microcomputer in 1980s
- world leading niche processors
- 2015 market share:
  - 85% of apps processors
  - 65% of computer peripherals,
  - 90% of hard-disk and SSD
  - 95% of automotive apps processors

ARM

- 32 bit RISC processor architecture family
- many variants
- ARM does not make chips
- licences the *IP core* (intellectual property) logic design to chip manufacturers

Books

- general reference :
  - W. Hohl, *ARM Assembly Language*, CRC Press, 2009
  - ARM7TDMI
  - NB not Linux format
- for Raspberry Pi:
  - B. Smith, *Raspberry Pi Assembly Language Raspbian Hands On Guide (2nd Ed)*, 2014

References

ARM Cortex A processor:

https://silver.arm.com/download/Software/BX100-DA-98001-r0p0-01rel3/DEN0013D_cortex_a_series_PG.pdf

General Purpose IO:

https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf

Memory

- linear addressing
- 2 descending stacks
  - main stack
  - process stack – used by OS
- all operations based on registers
- must move data between registers & memory

Registers

- 16 * 32 bit registers
- *working* registers
  - R0-R7 – *low* registers
  - R8-R12 – *high* registers
- R13 – stack pointer (SP)
- R14 - link register (LR)
- R15 – program counter (PC)
- PSR – program status register
  - bits for negative (N), zero (Z), carry (C), overflow (V)

Data formats

- *word*
  - 32 bits
  - `long`
- *half word*
  - 16 bits
  - `int`
- byte
  - 8 bits
  - `char`

Programming approach

- in high level programming:
  - identify variables & types
  - compiler maps these to memory & data representation
  - use of memory v registers invisible
- in low level programming
  - must make explicit use of memory & registers
  - must choose data representation
- registers much faster than memory
- try to map variables to registers
- what if more variables than registers...?

Instruction format

instruction Rd, Rn, operand$_2$

== Rd = Rn operation operand$_2$

- *instruction* is a mnemonic
  - meaningful short form
  - reminds us what instruction does
- *Rd* == destination register - `R0..R15`
- *Rn* == operand register - may be optional
- *operand$_2$ == register* or *constant*
- *constant* == #*number*

MOV: move

`MOV` *Rd, operand2* ➔

Rd = *operand2*

- don't update condition code flags

`MOVS` *Rd, operand2*

- as `MOV` but update condition code flags

`MOV` *Rd, #number* ➔

Rd = *number*

NB *number* <= 255 - more later

Label

identifier:

- can precede any line of assembler
- represents the address of:
  - instruction
  - literal data
- relative to program counter (PC)
- turned into PC + offset in machine code
- may start with _

System calls

- `SWI 0` == software interrupt to call function
- parameters in R0-R2
- system function number in R7
- e.g. `exit` is `1`

- NB shell variable `$?` is R0 so:

  `echo $?` == display R0

Layout

- usual to precede instruction with a *tab*
- comment is: @ *text*

Program skeleton

```
.global _start
_start:
@ program code here
_exit:
     MOV R0, #65 @ arbitrary value
     MOV R7, #1
     SWI 0
```

- `.global _start`
  - assembler directive
  - entry point `_start` is visible externally

Running assembler programs

- *file*`.s` == assembly language source

`$ as -o` *file*`.o` *file*`.s`

- assemble *file*`.s` to *file*`.o`

`$ ld -o` *file file*`.o`

- create executable *file* from *file*`.o`

`$ ./file`

- run executable in *file* from `_start`

`$ echo $?`

- display R0

Example: exit

- suppose `exit.s` is basic start program

```
$ as -o exit.o exit.s
$ ld -o exit exit.o
$ ./exit
$ echo $?
65
```

ADD: addition

`ADD` *Rd, Rn*, *operand*$_2$ ➔

Rd = *Rn*+*operand*$_2$

`ADD` *Rd, operand*$_2$ ➔

Rd = *Rd*+*operand*$_2$

`ADDC`

- add with carry
- like `ADD` + carry flag

`ADDS/ADDCS`

- like `ADD/ADDC` but set condition code flags

SUB: subtraction

`SUB` *Rd*, *Rn*, *operand*$_2$ ➔

Rd = *Rn*−*operand*$_2$

`SUB` *Rd*, *operand*$_2$ ➔

Rd = *Rd*−*operand*$_2$

`SUBC`

- subtract with carry
- like `SUB –1` if carry flag not set

`SUBS/SUBCS`

- like `SUB/SUBC` but set condition code flags

Multiply

- built in multiplication

`MUL` {*Rd*, } *Rn*, *operand*$_2$ ➔

Rd = *Rn* * *operand*$_2$

NB *Rd* cannot be an *operand*

Expressions

- for:

$var_1 = var_2\ op\ var_3$

- with:

var_1 in R_1

var_2 in R_2

var_3 in R_3

➔ $op\ R_1,R_2,R_3$

- e.g.

```
x = y*z; ➔

@ x == R1
@ y == R2
@ z == R3
MUL R1,R2,R3
```

Expressions

- for:

$var_1 = var_2\ op_1\ var_3\ op_2\ var_4$

- if op_1 and op_2 have same precedence
- or op_1 has higher precedence than op_2 ➔

$op_1\ R_1,R_2,R_3$

$op_2\ R_1,R_4$

e.g.

```
x = y*z-a; ➔
...
@ a == R4
MUL R1,R2,R3
SUB R1,R4
```

Expressions

- for:

$var_1 = var_2\ op_1\ var_3\ op_2\ var_4$

- if op_2 has higher precedence than op_1
- must evaluate *op2* expression in new register➔

$op_2\ R_i, R_3, R_4$

$op_1\ R_1, R_2, R_i$

e.g.

```
x = y+z*a; ➔
MUL R5,R3,R4
ADD R1,R2,R5
```

e.g.

```
x = y*(z+a) ➔
ADD R5,R3,R4
MUL R1,R2,R5
```

CMP: compare

`CMP` ***Rd*, *operand*$_2$**

- subtract *operand*$_2$ from *Rd* BUT ...
- ... do not modify *Rd*
  - otherwise same as `SUBS`
- update Z, N, C, V flags

`CMN` ***Rd*, *operand*$_2$**

- add *operand*$_2$ to *Rd* BUT ...
- ... do not modify *Rd*
  - otherwise same as `ADDS`
- update Z, N, C, V flags

Flags

- N – 1 if result <0; 0 otherwise
- Z – 1 if result =0; 0 otherwise
- C – 1 if result led to carry; 0 otherwise
  - i.e. $X+Y > 2^{32}$
  - i.e. X-Y >= 0

Flags

- V – 1 if result led to overflow; 0 otherwise
  - (-) == negative
  - (+) == positive
  - i.e. (-)X+(-)Y > 0
  - i.e. (+)X+(+)Y < 0
  - i.e. (-)X-(+)Y > 0
  - i.e. (+X)-(-Y) < 0

B: branch

`B` *label*

- branch to *label*
- i.e. reset PC to address for *label*

`B`*cond label*

- branch on *condition* to *label*

Condition

suffix	flags
`EQ` == equal	Z=1
`NE` == not equal	Z=0
`CS/HS` == carry set/ higher or same - unsigned	C=1
`CC/LO` == carry clear/lower - unsigned	C=0
`MI` == negative	N=1
`PL` == positive or 0	N=0
`VS` == overflow	V=1
`VC` == no overflow	V=0

Condition

suffix	flags
`HI` == higher - unsigned	C=1 & Z=0
`LS` == lower or same - unsigned	C=0 or Z=1
`GE` == greater than or equal - signed	N=V
`LT` == less than - signed	N!=V
`GT` == greater than - signed	Z=0 & N=V
`LE` == less than or equal, signed	Z=1 & N!=V
`AL` == any value	default if not *cond*

Example: multiply by adding

- m == x*y

```
int x;
int y;
int m;
x = 8;
y = 10;
m = 0;
while(y!=0)
{   m = m+x;
    y = y-1;
}
```

```
.global _start
_start:
   MOV R1, #0x03
   MOV R2, #0x04
   MOV R3, #0x00
_loop:
   CMP R2,#0x00
   BEQ _exit
   ADD R3, R1
   SUB R2, #0x01
   B  _loop
```

Example: multiply by adding

- m == x*y

```
int x;
int y;
int m;
x = 8;
y = 10;
m = 0;
while(y!=0)
{   m = m+x;
    y = y-1;
}
```

```
_exit:
  MOV R0, R3
  MOV R7, #1
  SWI 0
...
$ echo $?
12
```

Data directives & constants

`.data`

- start of sequence of data directives
- usually after instuctions program

`.equ` *label*`,` *value*

- define constant
- associate *label* with *value*
- use #*label* as $operand_2$ to get *value*

Example: multiply by adding

- m == x*y

```
int x;
int y;
int m;
x = 8;
y = 10;
m = 0;
while(y!=0)
{   m = m+x;
    y = y-1;
}
```

```
.global _start
_start:
   MOV R1, #X
   MOV R2, #Y
   MOV R3, #0x00
_loop:
   CMP R2,#0x00
   BEQ _exit
   ADD R3, R1
   SUB R2, #0x01
   B  _loop
```

Example: multiply by adding

- m == x*y

```
int x;
int y;
int m;
x = 8;
y = 10;
m = 0;
while(y!=0)
{   m = m+x;
    y = y-1;
}
```

```
_exit:
  MOV R0, R3
  MOV R7, #1
  SWI 0
.data
.equ X, 3
.equ Y, 4
...
$ echo $?
12
```

Tracing and debugging

- `gdb` - GNU debugger
- assemble with `-g` flag
- run program from `gdb`

```
$ as -g -o file.o file.s
$ ld -o file file.o
gdb file
...
(gdb) command
```

Tracing and debuggging

- `q(uit)` - return to shell
- `l(ist)` - show program - may need to press return
- `r(un)` - run program
- `b(reak)` *number* – set breakpoint at line *number*
  - program pauses at breakpoint
  - can set multiple breakpoints
  - can also set break points at labels & addresses

Tracing and debuggging

- `i(nfo) r` – show registers
- `c(ontinue)` - continue execution after breakpoint
- `s(tep)` - execute next instruction
- `i(nfo) b` - show breakpoints
- `d(elete)` *number* – remove breakpoint *number*

Tracing and debugging

- to trace
  - set breakpoint at start of program
  - step & show registers
- to debug
  - set breakpoints at salient points in program
  - e.g. at start or ends of loops
  - continue/step & show registers

Tracing and debugging

- e.g. suppose mult program is in `mult.s`

```
$ as -g -o mult.o mult.s
$ ld -o mult mult.s
$ gdb mult
...
(gdb) l
1       .global _start
2
3 _start:
4           MOV R1, #X
5           MOV R2, #Y
6           MOV R3, #0x00
7 _loop:
8           CMP R2,#0x00
9           BEQ _exit
10          ADD R3, R1
(gdb)
11          SUB R2, #0x01
12          B   _loop
13 _exit:
14          MOV R0, R3
15          MOV R7, #1
16          SWI 0
17
18 .data
19 .equ X, 3
20 .equ Y, 4
(gdb)
```

Tracing & debugging

```
(gdb) b _start
Breakpoint 1 at 0x805b:
file mult.s, line 5
(gdb) r
...
5         MOV R2, #Y
(gdb) i r
r0      0x0     0
r1      0x3     3
r2      0x0     0
r3      0x0     0
(gdb) s
6         MOV R, #0
```

```
(gdb) i r
r0      0x0     0
r1      0x3     3
r2      0x0     4
r3      0x0     0
...
(gdb) b 12
Breakpoint 2 at 0x8070:
file mult.s, line 12
(gdb) c
...
12        B _loop
```

Tracing & debugging

```
(gdb) i r
r0      0x0     0
r1      0x3     3
r2      0x0     3
r3      0x0     3
...
(gdb) c
...
12        B _loop
(gdb) i r
r0      0x0     0
r1      0x3     3
r2      0x0     2
r3      0x0     6
...
```

```
(gdb) c
...
12        B _loop
(gdb) i r
r0      0x0     0
r1      0x3     3
r2      0x0     1
r3      0x0     9
...
(gdb) c
...
12        B _ loop
```

Tracing & debugging

```
(gdb) i r
r0      0x0    0
r1      0x3    3
r2      0x0    0
r3      0x0    12
...
(gdb) c
...
[Inferior 1 (process 2614)
exited with code 014]
(gdb) q
$
```

NOP: no operation

`NOP`

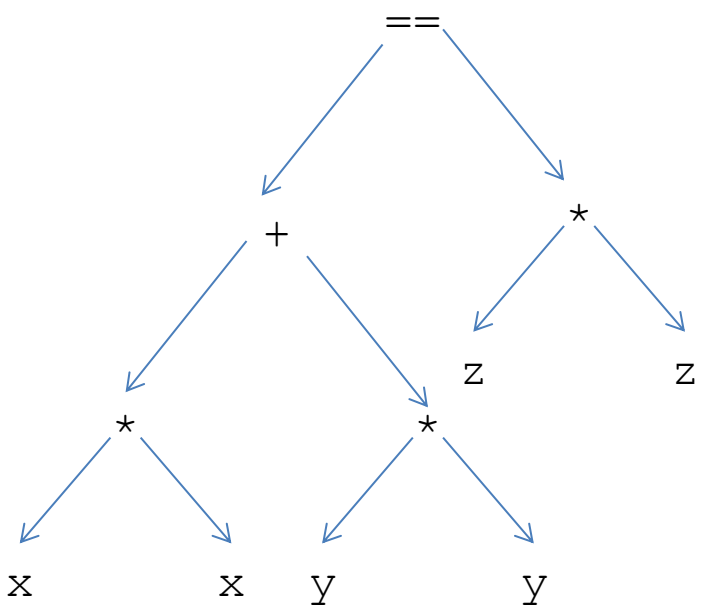
- do nothing
- use for padding/layout
- no cost

Expressions

- draw expression tree
- allocate registers to nodes
  - from bottom left
  - if expression in assignment then start with register for destination variable
- accumulate into register for left operand
- can re-use any register whose value is no longer required

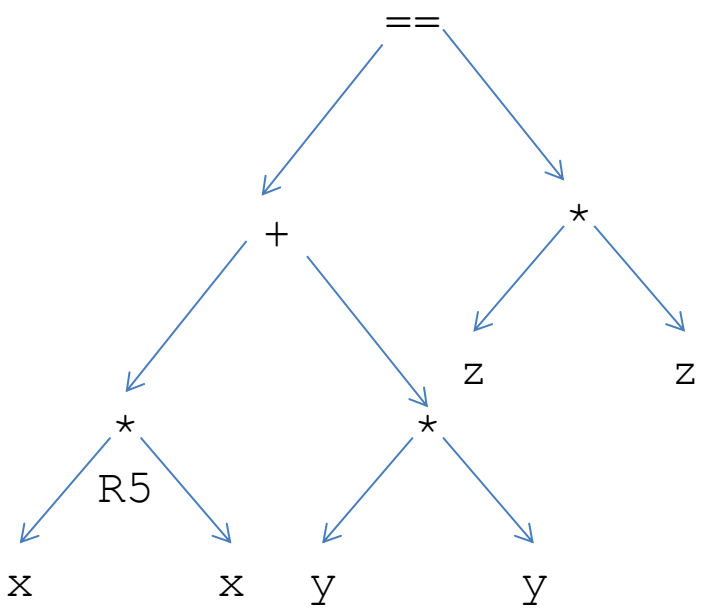
Example: Pythagoras

```
int x; /* R1 */
int y; /* R2 */
int z; /* R3 */
int p; /* R4 */
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```



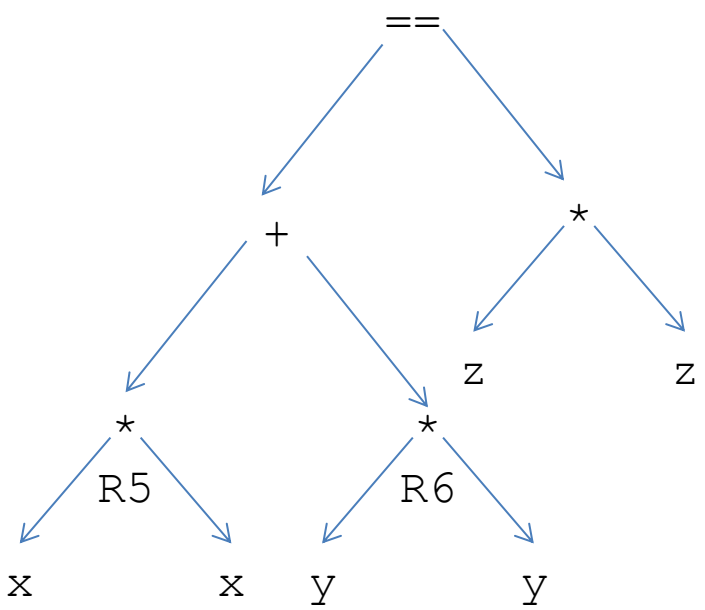
Example: Pythagoras

```
int x; /* R1 */
int y; /* R2 */
int z; /* R3 */
int p; /* R4 */
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```



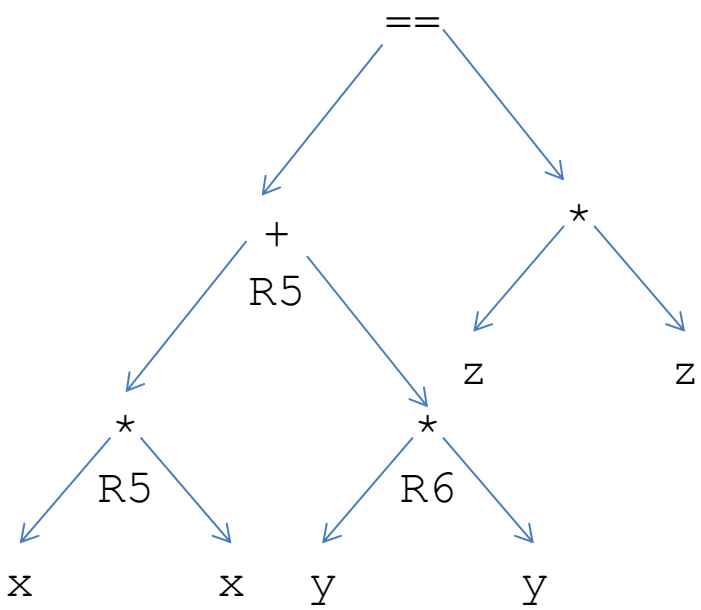
Example: Pythagoras

```
int x; /* R1 */
int y; /* R2 */
int z; /* R3 */
int p; /* R4 */
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```



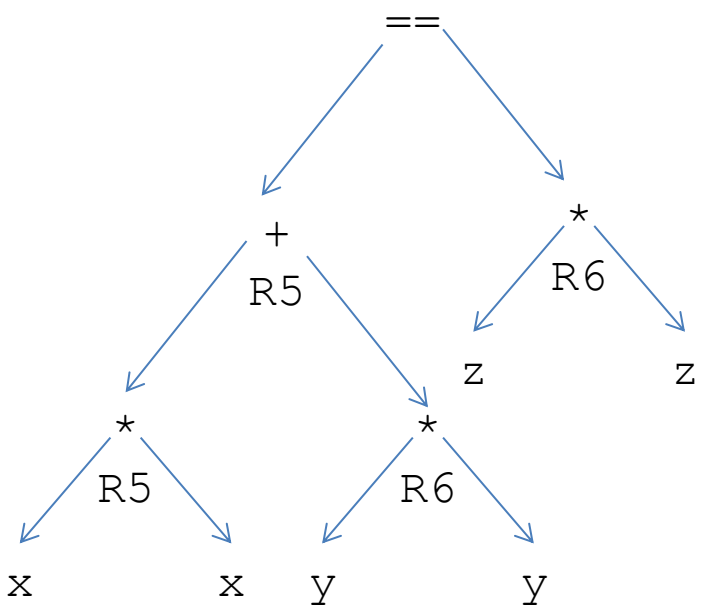
Example: Pythagoras

```
int x; /* R1 */
int y; /* R2 */
int z; /* R3 */
int p; /* R4 */
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```



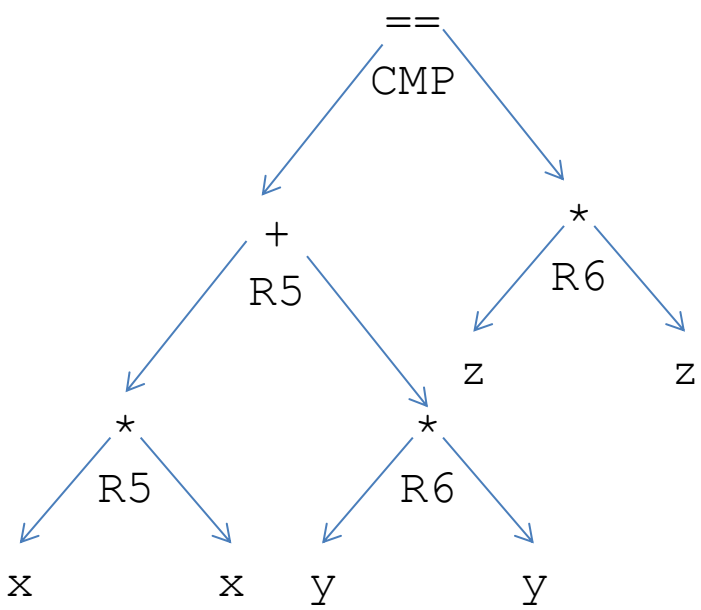
Example: Pythagoras

```
int x; /* R1 */
int y; /* R2 */
int z; /* R3 */
int p; /* R4 */
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```



Example: Pythagoras

```
int x; /* R1 */
int y; /* R2 */
int z; /* R3 */
int p; /* R4 */
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```



Example: Pythagoras

```
int x;
int y;
int z;
int p;
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```

```
.global _start
_start:
        MOV R1, #X
        MOV R2, #Y
        MOV R3, #Z
        MUL R5, R1, R1
        MUL R6, R2, R2
        ADD R5, R6
        MUL R6, R3, R3
        CMP R5, R6
        BEQ _same
        MOV R4, #0
        B _exit
_same:
        MOV R4, #1
```

Example: Pythagoras

```
int x;
int y;
int z;
int p;
x = 3;
y = 4;
z = 5;
if(x*x+y*y==z*z)
 p=1;
else
 p=0;
```

```
_exit:
        MOV R0,  R4
        MOV R7,  #1
        SWI 0
.data
.equ X,  3
.equ Y,  4
.equ Z,  5
```