Princeton University
COS 217: Introduction to Programming Systems
ARMv8 Architecture

Memory Chip

CPU Chip

Control Unit

Arithmetic Logic Unit (ALU)

Registers

r0

r1

r2

...

r28

r29

r30

zr

sp

pc

pstate

...

0000000000000000

TEXT

RODATA

DATA

BSS

HEAP

The
Program
Break

argc

argv

...

argv array

env var array

...

argv strings

env var strings

STACK

FFFFFFFFFFFFFFFF
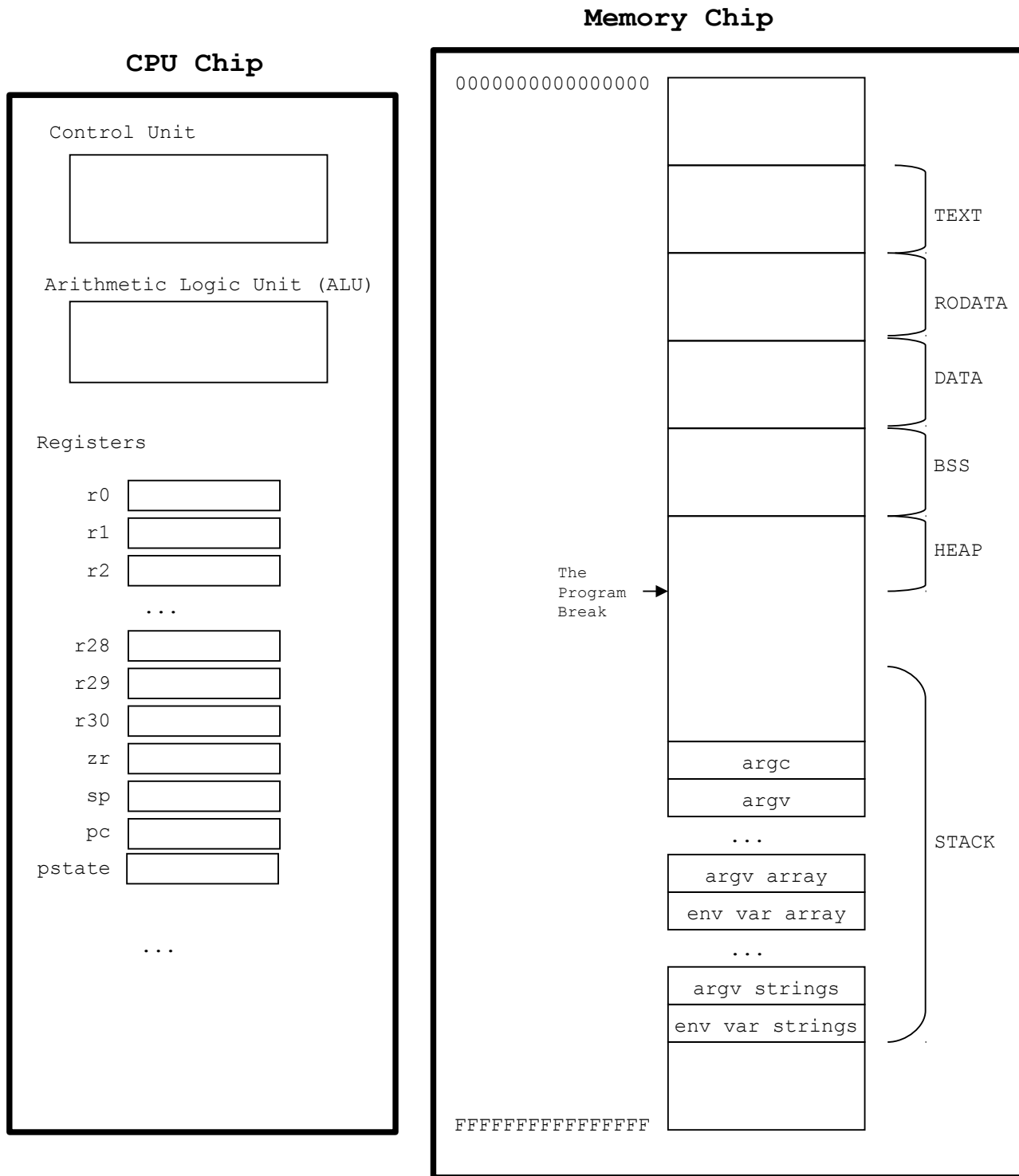
Princeton University
COS 217: Introduction to Programming Systems
ARMv8 Registers

General Registers

Name	Bits 63-0	Bits 31-0	Description	Call Convention
r0	x0	w0	Argument 0, scratch, return value	caller-saved
r1	x1	w1	Argument 1, scratch	caller-saved
r2	x2	w2	Argument 2, scratch	caller-saved
r3	x3	w3	Argument 3, scratch	caller-saved
r4	x4	w4	Argument 4, scratch	caller-saved
r5	x5	w5	Argument 5, scratch	caller-saved
r6	x6	w6	Argument 6, scratch	caller-saved
r7	x7	w7	Argument 7, scratch	caller-saved
r8	x8	w8	Indirect result location (XR)	
r9	x9	w9	Scratch	caller-saved
r10	x10	w10	Scratch	caller-saved
r11	x11	w11	Scratch	caller-saved
r12	x12	w12	Scratch	caller-saved
r13	x13	w13	Scratch	caller-saved
r14	x14	w14	Scratch	caller-saved
r15	x15	w15	Scratch	caller-saved
r16	x16	w16	Intra-procedure call (IP0)	
r17	x17	w17	Intra-procedure call (IP1)	
r18	x18	w18	Platform register (PR)	
r19	x19	w19	Local variable	callee-saved
r20	x20	w20	Local variable	callee-saved
r21	x21	w21	Local variable	callee-saved
r22	x22	w22	Local variable	callee-saved
r23	x23	w23	Local variable	callee-saved
r24	x24	w24	Local variable	callee-saved
r25	x25	w25	Local variable	callee-saved
r26	x26	w26	Local variable	callee-saved
r27	x27	w27	Local variable	callee-saved
r28	x28	w28	Local variable	callee-saved
r29	x29	w29	Frame pointer (FP)	
r30	x30	w30	Procedure link register (LR)	

Special Registers

Name	Bits 63-0	Bits 31-0	Description
zr	xzr	wzr	Zero register
sp	sp	wsp	Stack pointer
pc	pc		Program counter
pstate		pstate	Processor state; contains the N, Z, C, and V condition flags

Princeton University
COS 217: Introduction to Programming Systems
A Subset of ARMv8 Assembly Language

Simplifying assumptions: We will consider only programs whose functions:
- do not use floating point values,
- have parameters that are integers or addresses (but not structures),
- have return values that are integers or addresses (but not structures), and
- have no more than 8 parameters.

Comments

```
// This is a comment
```

Label Definitions

symbol:
> Record the fact that *symbol* is a label that marks the current location within the current section

Directives

`.section .`*sectionname*
> Make the *sectionname* section the current section; *sectionname* may be `text`, `rodata`, `data`, or `bss`

`.size` *symbol*, *expr*
> Set the size associated with *symbol* to the value of expression *expr*

`.skip` *n*
> Skip *n* bytes of memory in the current section

`.byte` *value1*, *value2*, `...`
> Allocate one byte of memory containing *value1*, one byte of memory containing *value2*, ... in the current section

`.short` *value1*, *value2*, …
> Allocate two bytes (a half word) of memory containing *value1*, two bytes (a half word) of memory containing *value2*, ... in the current section

`.word` *value1*, *value2*, …
> Allocate four bytes (a word) of memory containing *value1*, four bytes (a word) of memory containing *value2*, ... in the current section

`.quad` *value1*, *value2*, …
> Allocate eight bytes (an extended word) of memory containing value1, eight bytes (an extended word) of memory containing value2, ... in the current section

`.ascii "`*string1*`", "`*string2*`"`, …
> Allocate memory containing the characters from *string1*, *string2*, ... in the current section

`.string "`*string1*`", "`*string2*`"`, …
> Allocate memory containing *string1*, *string2*, ..., where each string is `'\0'` terminated, in the current section

`.equ` *symbol*, *expr*
> Define *symbol* to be an alias for the value of expression *expr*

symbol `.req` *reg*
> Define *symbol* to be an alias for register *reg*

Instructions

The following is a subset and simplification of information provided in the manual *ARMv8 Instruction Set Overview*.

Key

`Wn`	4 byte general register, or WZR
`Wn|WSP`	4 byte general register, or WSP
`Xn`	8 byte general register, or XZR
`Xn|SP`	8 byte general register, or SP
`imm`	Immediate operand, that is, an integer
`addr`	Memory address having one of these forms:

```
[Xn]
[Xn, imm]
[Xn, Xm]
[Xn, Xm, lsl 1]   where the loaded/stored object consists of 2 bytes
[Xn, Xm, lsl 2]   where the loaded/stored object consists of 4 bytes
[Xn, Xm, lsl 3]   where the loaded/stored object consists of 8 bytes
```

Data Copy Instructions

```
MOV Wd, imm
      Wd = imm
MOV Xd, imm
      Xd = imm
MOV Wd|WSP, Ws|WSP
      Wd|WSP = Ws|WSP
MOV Xd|SP, Xs|SP
      Xd|SP = Xs|SP
```

Address Generation Instruction

```
ADR Xd, symbol
      Place in Xd the address denoted by label symbol
```

Memory Access Instructions

```
LDR Wd, addr
      Load 4 bytes from memory addressed by addr to Wd
LDR Xd, addr
      Load 8 bytes from memory addressed by addr to Xd
LDRB Wd, addr
      Load 1 byte from memory addressed by addr, then zero-extend it to Wd
LDRSB Wd, addr
      Load 1 byte from memory addressed by addr, then sign-extend it into Wd
LDRSB Xd, addr
      Load 1 byte from memory addressed by addr, then sign-extend it into Xd
LDRH Wd, addr
      Load 2 bytes from memory addressed by addr, then zero-extend it into Wd
LDRSH Wd, addr
      Load 2 bytes from memory addressed by addr, then sign-extend it into Wd
LDRSH Xd, addr
      Load 2 bytes from memory addressed by addr, then sign-extend it into Xd
```

```
LDRSW Xd, addr
```
    Load 4 bytes from memory addressed by `addr`, then sign-extend it into `Xd`
```
STR Ws, addr
```
    Store 4 bytes from `Ws` to memory addressed by `addr`
```
STR Xs, addr
```
    Store 8 bytes from `Xs` to memory addressed by `addr`
```
STRB Ws, addr
```
    Store 1 bytes from `Ws` to memory addressed by `addr`
```
STRH Ws, addr
```
    Store 2 byes from `Ws` to memory addressed by `addr`

Arithmetic Instructions

```
ADD Wd|WSP, Ws|WSP, imm
```
    `Wd|WSP = Ws|WSP + imm`
```
ADD Xd|SP, Xs|SP, imm
```
    `Xd|SP = Xs|SP + imm`
```
ADD Wd|WSP, Ws|WSP, Wm
```
    `Wd|WSP = Ws|WSP + Wm`
```
ADD Xd|SP, Xs|SP, Wm
```
    `Xd|SP = Xs|SP + Wm`
```
ADD Xd|SP, Xs|SP, Xm
```
    `Xd|SP = Xs|SP + Xm`
```
ADDS Wd, Ws|WSP, imm
```
    `Wd = Ws|WSP + imm`, setting each condition flag to 0 or 1 based upon the result
```
ADDS Xd, Xs|SP, imm
```
    `Xd = Xs|SP + imm`, setting each condition flag to 0 or 1 based upon the result
```
ADDS Wd, Ws|WSP, Wm
```
    `Wd = Ws|WSP + Wm`, setting each condition flag to 0 or 1 based upon the result
```
ADDS Xd, Xs|SP, Wm
```
    `Xd = Xs|SP + Wm`, setting each condition flag to 0 or 1 based upon the result
```
ADDS Xd, Xs|SP, Xm
```
    `Xd = Xs|SP + Xm`, setting each condition flag to 0 or 1 based upon the result
```
ADC Wd, Ws, Wm
```
    `Wd = Ws + Wm + C`
```
ADC Xd, Xs, Xm
```
    `Xd = Xs + Xm + C`
```
ADCS Wd, Ws, Wm
```
    `Wd = Ws + Wm + C`, setting each condition flag to 0 or 1 based upon the result
```
ADCS Xd, Xs, Xm
```
    `Xd = Xs + Xm + C`, setting each condition flag to 0 or 1 based upon the result
```
SUB Wd|WSP, Ws|WSP, imm
```
    `Wd|WSP = Ws|WSP - imm`
```
SUB Xd|SP, Xs|SP, imm
```
    `Xd|SP = Xs|SP - imm`
```
SUB Wd|WSP, Ws|WSP, Wm
```
    `Wd|WSP = Ws|WSP - Wm`
```
SUB Xd|SP, Xs|SP, Wm
```
    `Xd|SP = Xs|SP - Wm`
```
SUB Xd|SP, Xs|SP, Xm
```
    `Xd|SP = Xs|SP - Xm`
```
SUBS Wd, Ws|WSP, imm
```
    `Wd = Ws|WSP - imm`, setting each condition flag to 0 or 1 based upon the result
```
SUBS Xd, Xs|SP, imm
```
    `Xd = Xs|SP - imm`, setting each condition flag to 0 or 1 based upon the result

```
SUBS Wd, Ws|WSP, Wm
      Wd = Ws|WSP - Wm, setting each condition flag to 0 or 1 based upon the result
SUBS Xd, Xs|SP, Wm
      Xd = Xs|SP - Wm, setting each condition flag to 0 or 1 based upon the result
SUBS Xd, Xs|SP, Xm
      Xd = Xs|SP - Xm, setting each condition flag to 0 or 1 based upon the result
MUL Wd, Ws, Wm
      Wd = Ws * Wm
MUL Xd, Xs, Xm
      Xd = Xs * Xm
SDIV Wd, Ws, Wm
      Wd = Ws / Wm, treating source operands as signed
SDIV Xd, Xs, Xm
      Xd = Xs / Xm, treating source operands as signed
UDIV Wd, Ws, Wm
      Wd = Ws / Wm, treating source operands as unsigned
UDIV Xd, Xs, Xm
      Xd = Xs / Xm, treating source operands as unsigned
```

Logical Instructions

```
MVN Wd, Ws
      Wd = ~Ws
MVN Xd, Xs
      Xd = ~Xs
AND Wd|WSP, Ws, imm
      Wd|WSP = Ws & imm
AND Xd|SP, Xs, imm
      Xd|SP = Xs & imm
AND Wd, Ws, Wm
      Wd = Ws & Wm
AND Xd, Xs, Xm
      Xd = Xs & Xm
ANDS Wd, Ws, imm
      Wd = Ws & imm, setting condition flag N to 0 or 1 based upon the result, Z to 0 or 1 based
      upon the result, C to 0, and V to 0
ANDS Xd, Xs, imm
      Xd = Xs & imm, setting condition flag N to 0 or 1 based upon the result, Z to 0 or 1 based
      upon the result, C to 0, and V to 0
ANDS Wd, Ws, Wm
      Wd = Ws & Wm, setting condition flag N to 0 or 1 based upon the result, Z to 0 or 1 based upon
      the result, C to 0, and V to 0
ANDS Xd, Xs, Xm
      Xd = Xs & Xm, setting condition flag N to 0 or 1 based upon the result, Z to 0 or 1 based upon
      the result, C to 0, and V to 0
ORR Wd|WSP, Ws, imm
      Wd|WSP = Ws | imm
ORR Xd|SP, Xs, imm
      Xd|SP = Xs | imm
ORR Wd, Ws, Wm
      Wd = Ws | Wm
ORR Xd, Xs, Xm
      Xd = Xs | Xm
EOR Wd|WSP, Ws, imm
      Wd|WSP = Ws ^ imm
EOR Xd|SP, Xs, imm
```

```
        Xd|SP = Xs ^ imm
EOR Wd, Ws, Wm
        Wd = Ws ^ Wm
EOR Xd, Xs, Xm
        Xd = Xs ^ Xm
```

Shift Instructions

```
LSL Wd, Ws, imm
        Wd = Ws << imm
LSL Xd, Xs, imm
        Xd = Xs << imm
LSL Wd, Ws, Wm
        Wd = Ws << Wm
LSL Xd, Xs, Xm
        Xd = Xs << Xm
LSR Wd, Ws, imm
        Wd = Ws >> imm (logical shift)
LSR Xd, Xs, imm
        Xd = Xs >> imm (logical shift)
LSR Wd, Ws, Wm
        Wd = Ws >> Wm (logical shift)
LSR Xd, Xs, Xm
        Xd = Xs >> Xm (logical shift)
ASR Wd, Ws, imm
        Wd = Ws >> imm (arithmetic shift)
ASR Xd, Xs, imm
        Xd = Xs >> imm (arithmetic shift)
ASR Wd, Ws, Wm
        Wd = Ws >> Wm (arithmetic shift)
ASR Xd, Xs, Xm
        Xd = Xs >> Xm (arithmetic shift)
```

Branch Instructions

```
CMP Ws|WSP, imm
        Alias for SUBS WZR, Ws|WSP, imm
CMP Xs|SP, imm
        Alias for SUBS XZR, Xs|SP, imm
CMP Ws|WSP, Wm
        Alias for SUBS WZR, Ws|WSP, Wm
CMP Xs|SP, Wm
        Alias for SUBS XZR, Xs|SP, Wm
CMP Xs|SP, Xm
        Alias for SUBS XZR, Xs|SP, Xm
B symbol
        Jump to label symbol
Bcond symbol
        Jump to label symbol if and only if cond is true, where cond is defined by this table:
```

Cond	Meaning	Condition Flags
EQ	Equal	Z==1
NE	Not equal	Z==0

```
LT    Signed less than           N!=V
LE    Signed less than or equal  N!=V || Z==1
GT    Signed greater than        N==V && Z==0
GE    Signed greater than or equal N==V

LO    Unsigned lower             C==0
LS    Unsigned lower or same     C==0 || Z==1
HI    Unsigned higher            C==1 && Z==0
HS    Unsigned higher or same    C==1

MI    Minus (negative)           N==1
PL    Plus (positive or 0)       N==0

VS    Overflow set               V==1
VC    Overflow clear             V==0

CS    Carry set                  C==1
CC    Carry clear                C==0
```

CBNZ *Ws*, *symbol*
> Jump to label *symbol* if and only if *Ws* is not equal to zero

CBNZ *Xs*, *symbol*
> Jump to label *symbol* if and only if *Xs* is not equal to zero

CBZ *Ws*, *symbol*
> Jump to label *symbol* if and only if *Ws* is equal to zero

CBZ *Xs*, *symbol*
> Jump to label *symbol* if and only if *Xs* is equal to zero

Function Call/Return Instructions

BL *symbol*
> Place the address of the next sequential instruction in register X30, and jump to label *symbol*

RET
> Jump to the instruction which is at the address in register X30

```
 1: /*------------------------------------------------------------------*/
 2: /* hello.c                                                          */
 3: /* Author: Bob Dondero                                              */
 4: /*------------------------------------------------------------------*/
 5:
 6: #include <stdio.h>
 7:
 8: /*------------------------------------------------------------------*/
 9:
10: /* Write "hello, world\n" to stdout.  Return 0. */
11:
12: int main(void)
13: {
14:    printf("hello, world\n");
15:    return 0;
16: }
```

hello.c (Page 1 of 1)

```
 1: /*------------------------------------------------------------------*/
 2: /* hello.c                                                          */
 3: /* Author: Bob Dondero                                              */
```

```
 1: //-------------------------------------------------------------------
 2: // hello.s
 3: // Author: Bob Dondero and William Ughetta
 4: //-------------------------------------------------------------------
 5:
 6:         .section .rodata
 7:
 8: greetingStr:
 9:         .string "hello, world\n"
10:
11: //-------------------------------------------------------------------
12:
13:         .section .data
14:
15: //-------------------------------------------------------------------
16:
17:         .section .bss
18:
19: //-------------------------------------------------------------------
20:
21:         .section .text
22:
23:         //-----------------------------------------------------------
24:         // Write "hello, world\n" to stdout.  Return 0.
25:         // int main(void)
26:         //-----------------------------------------------------------
27:
28:         // Must be a multiple of 16
29:         .equ    MAIN_STACK_BYTECOUNT, 16
30:
31:         .global main
32:
33: main:
34:
35:         // Prolog
36:         sub     sp, sp, MAIN_STACK_BYTECOUNT
37:         str     x30, [sp]
38:
39:         // printf("hello, world\n")
40:         adr     x0, greetingStr
41:         bl      printf
42:
43:         // Epilog and return 0
44:         mov     w0, 0
45:         ldr     x30, [sp]
46:         add     sp, sp, MAIN_STACK_BYTECOUNT
47:         ret
48:
49:         .size   main, (. - main)
```