PES UNIVERSITY ONLINE

Microprocessor & Computer Architecture (µpCA)

Introduction to ARM

Dr. D. C. Kiran
Department of Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Syllabus

Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- Introduction to ARM Processor & Applications
- ARM Architecture Overview
- Different ARM processor Modes

Unit 2: Pipelined Processor and Design

Unit 3: Memory Design

Unit 4: Input/Output Device Design

Unit 5: Advanced Architecture

ARM: [ACORN RISC MACHINE]

- Leading provider of 32-bit embedded RISC microprocessors, 75% of market
  - High performance
  - Low power consumption
  - Low system cost
- Solutions for
  - Embedded real-time systems for mass storage, automotive, industrial and networking applications
  - Secure applications - smartcards and SIMs
  - Open platforms running complex operating systems

Microprocessor & Computer Architecture (µpCA)

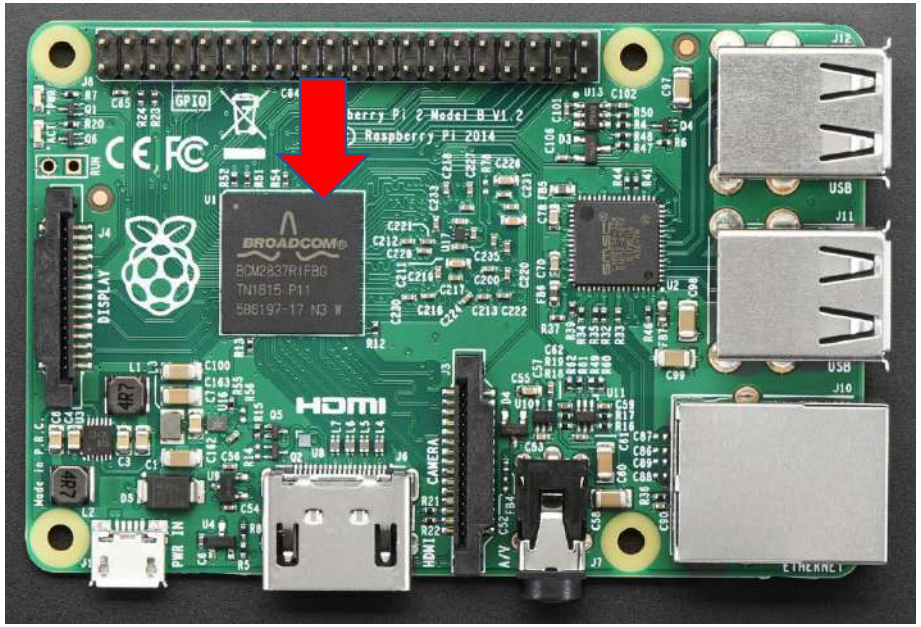
ARM Processor

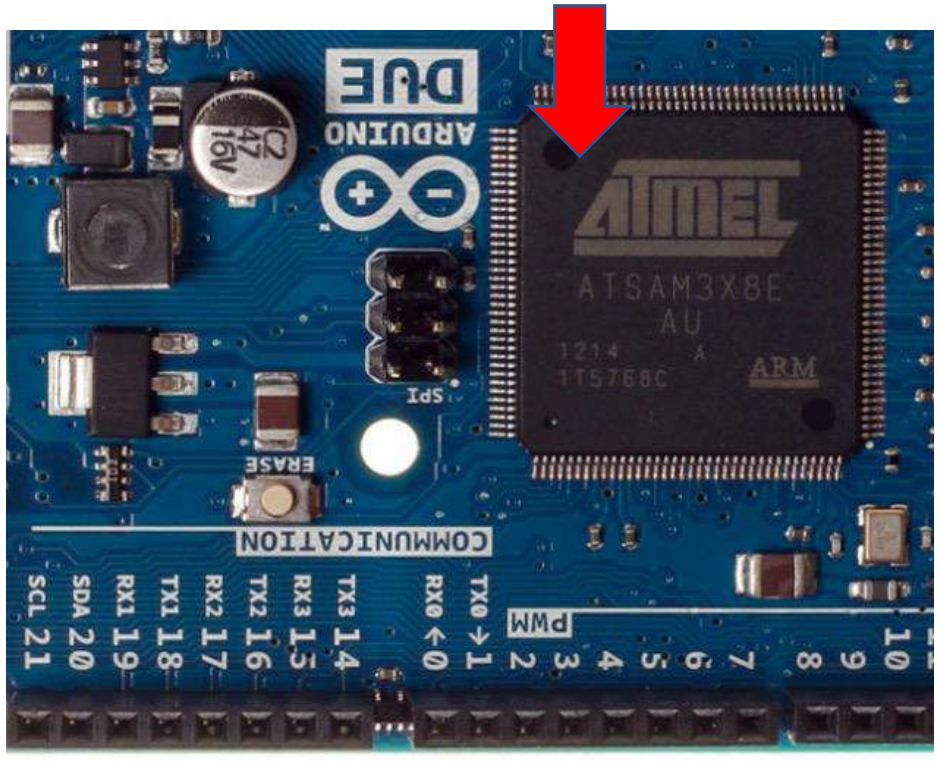
Microprocessor & Computer Architecture (µpCA)

Microprocessor & Computer Architecture (µpCA)

1: The **Raspberry Pi** 4 1.5 GHz 64-bit quad-core **ARM** Cortex-A72 (BCM2837)processor.

2: The **Arduino Due** : Atmel SAM3X8E ARM Cortex-M3 CPU.





Microprocessor & Computer Architecture (µpCA)

- ARM Based Mine Detection Robot Using GPS Technology
- ARM Based Hospital Enquiry System
- Temperature and Humidity Control System Using ARM and Graphical LCD
- Multi Functional Car With Accident Alert Sensors Using ARM
- ARM Based Vehicle Tracking System Using GPS and GSM
- Biometric fingerprint Identification based Bank Locker Security System Using ARM
- ARM Base Automated Bus Arrival Announcement System for the Blind Persons.
- ARM Based Digital Notice Board Using GSM

The ARM Architecture Versions

- **Version 1 (ARM1):** 26 bit addressing, no coprocessor.
- **Version 2 (ARM2):** Includes a 32 bit result multiply coprocessor
- **Version 2as (ARM3 & 250):**
- **Version 3 (ARM6 ,7,8):** 32 bit addressing
- **Version 4 (Strong ARM, ARM9):** Half word load/store instructions were provided.
- **Version 4T:** Thumbing: 16 bit instructions can be compressed in a 32 bit processor, thus enabling more instructions to be packed in the same memory, thereby increasing the code density.
- **Version 5T and 5TE (ARM10):** 5TE: thumb extension- built for powerful computations.
- CORTEX-M, CORTEX-R, CORTEX-A (32 Bit and 64 bit), NEOVERSE

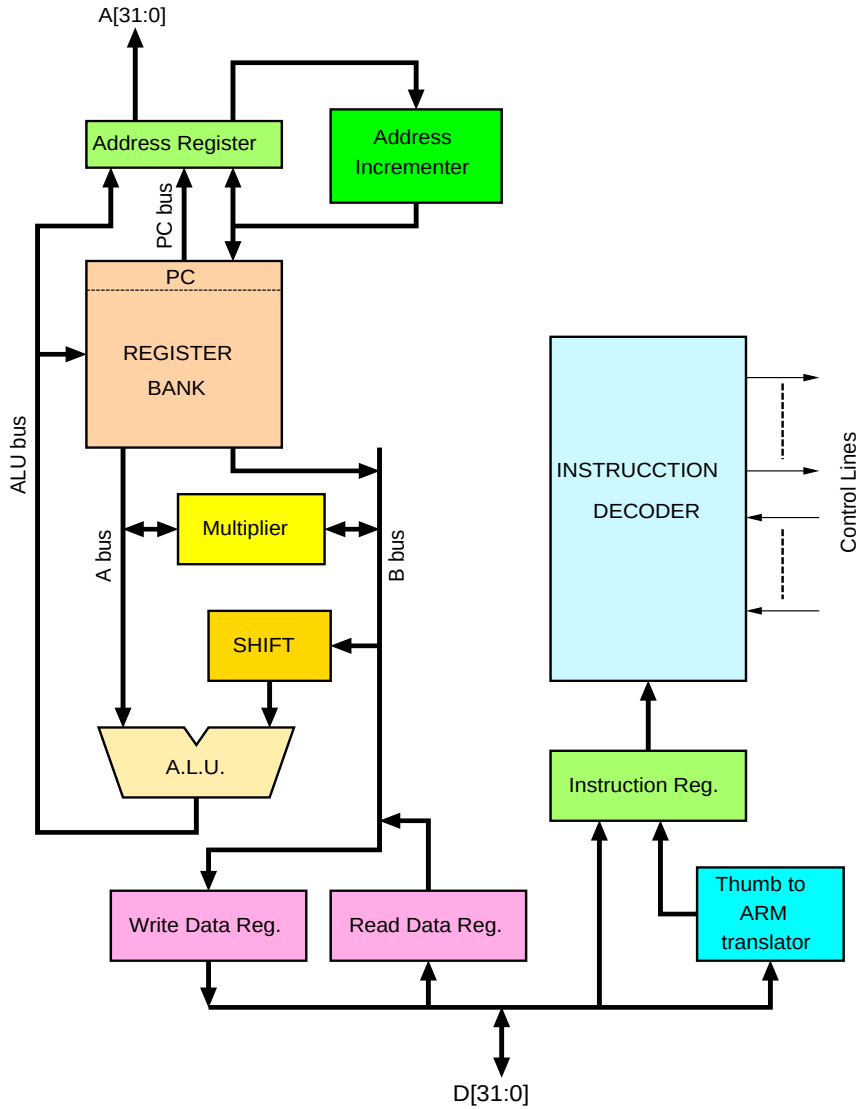
Reference
https://www.cs.umd.edu/~meesh/cmsc411/website/proj01/arm/armchip.html

ARM7TDMI Processor

- Current low-end ARM core for applications like digital mobile phones
- TDMI
  - **T**: Thumb, 16-bit instruction set
  - **D**: on-chip Debug support, enabling the processor to halt in response to a debug request
  - **M**: enhanced Multiplier, yield a full 64-bit result, high performance
  - **I**: EmbeddedICE hardware
- Von Neumann architecture
- 3-stage pipeline

Microprocessor & Computer Architecture (µpCA)

- Von Neumann Architecture
- 3-Stage Pipeline
  - Fetch, Decode, Execute
- 32-bit Data bus
- 32-bit Address Bus
- 37 32-bit Registers
- 32-bit ARM Instruction Set
- 16-bit THUMB instruction Set
- 32x8 Multiplier
- Barrel Shifter

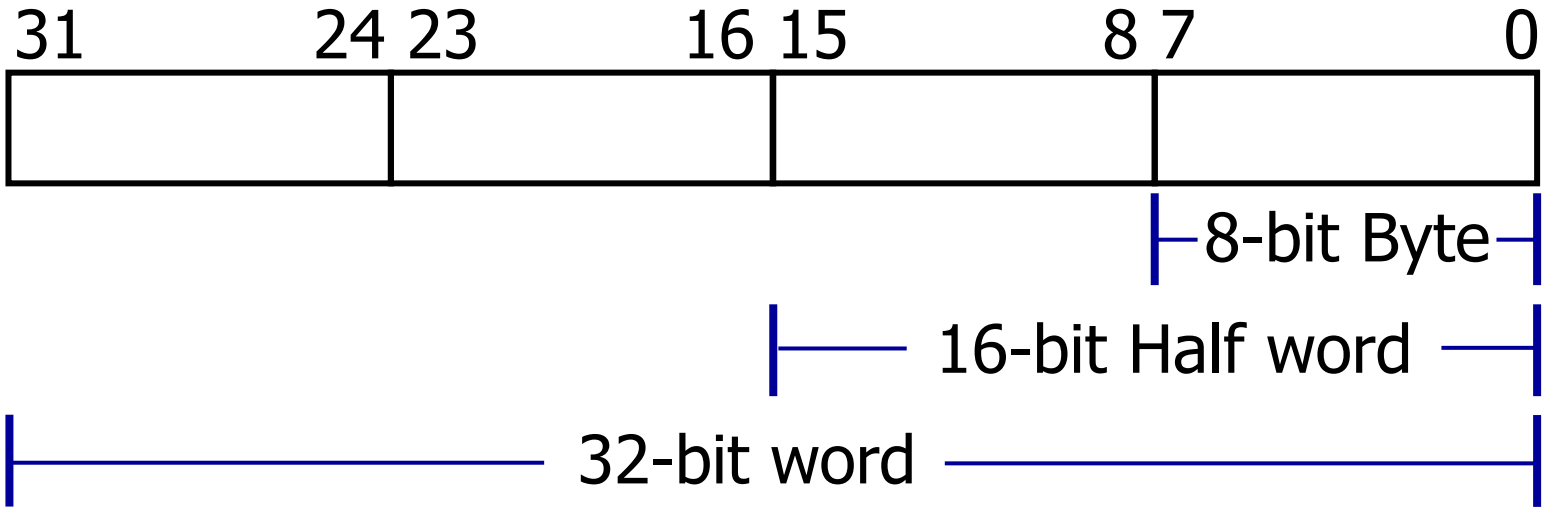


Data Sizes and Instruction Sets

- The ARM is a 32-bit architecture.

- When used in relation to the ARM:
  - **Byte** means 8 bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)

- Most ARM's implement two instruction sets
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set



ARM processors support six data types.

- **8-bit** signed and unsigned bytes.
- **16-bit** signed and unsigned half-words; these are aligned on 2-byte boundaries.
- **32-bit** signed and unsigned words; these are aligned on 4-byte boundaries.

Processor Modes

The ARM has seven basic operating modes:

User

FIQ (High Priority)

IRQ (Low Priority)

Supervisor

Abort

Undefined

System

Modes other than User mode are collectively known as Privileged modes. Privileged modes are used to service interrupts or exceptions, or to access protected resources.

What happens when Program is Executing?

Program may Complete its execution

Or

Execution may be halted Temporarily or Permanently

Why Program Execution will be halted Temporarily?

Function / Subroutine Call (User Program)

Or

Operating System want to perform few maintenance work

Or

Error Handling Operation to be Executed, (Also called Interrupt Handling)

User Program will have lowest priority

Context switch with System Program and User Program

System Programs may have same or different priority

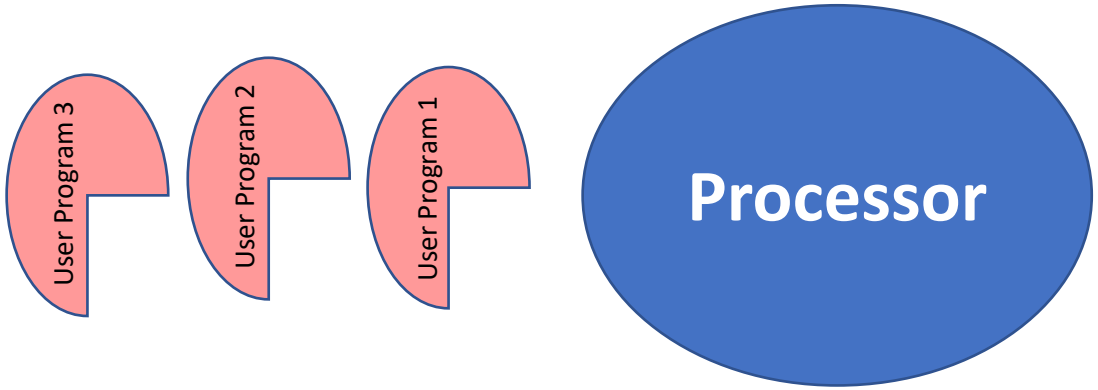
If Higher Priority program interrupts, Lower Priority Program will be switched.

If same priority program interrupts, no context switch take place.

During Context switching data related to program should be saved

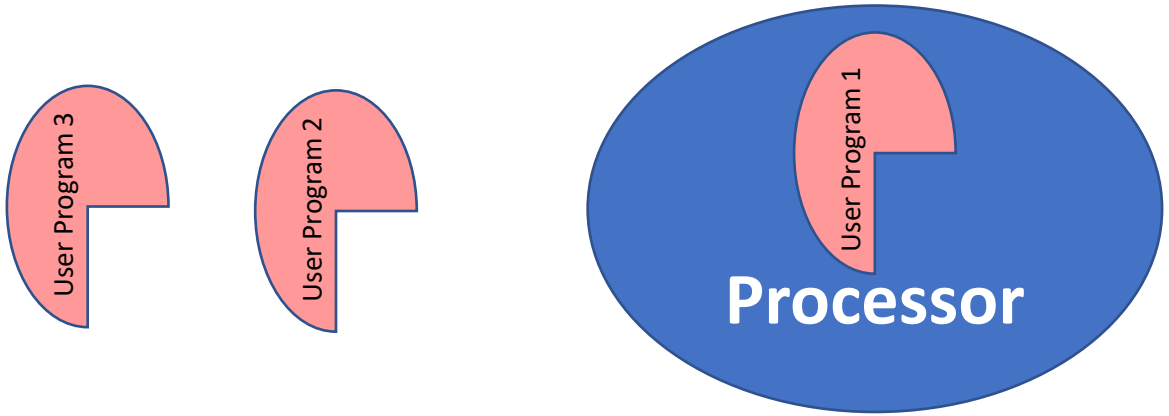
During Return, saved date should be restored back to program.

Processor Modes



User Program 1, User Program 2 and User Program 3 are waiting for CPU time for execution

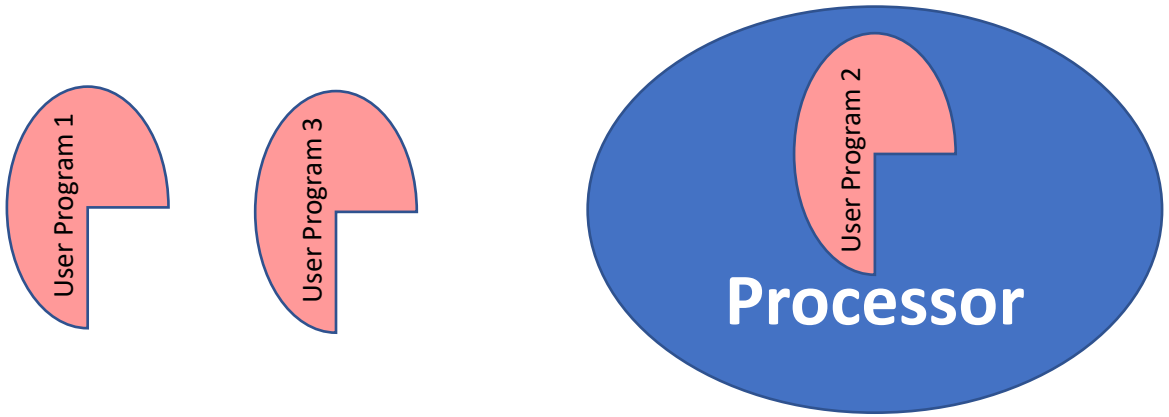
Processor Modes



User Program 1 is Executing

User Program 2 and User Program 3 are waiting for CPU time for execution

Processor Modes

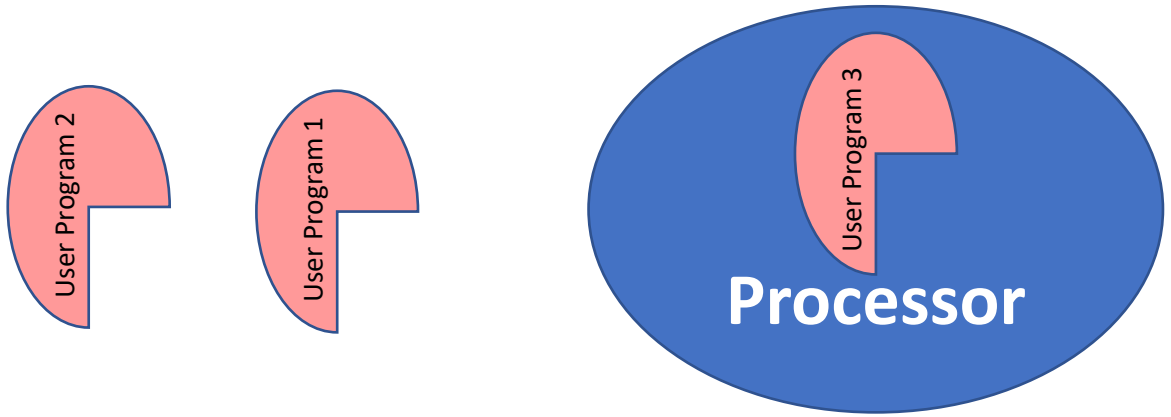


Context Switching between User Program 1 and User Program 2

User Program 2 is Executing

User Program 3 and User Program 1 are waiting for CPU time for execution

Processor Modes



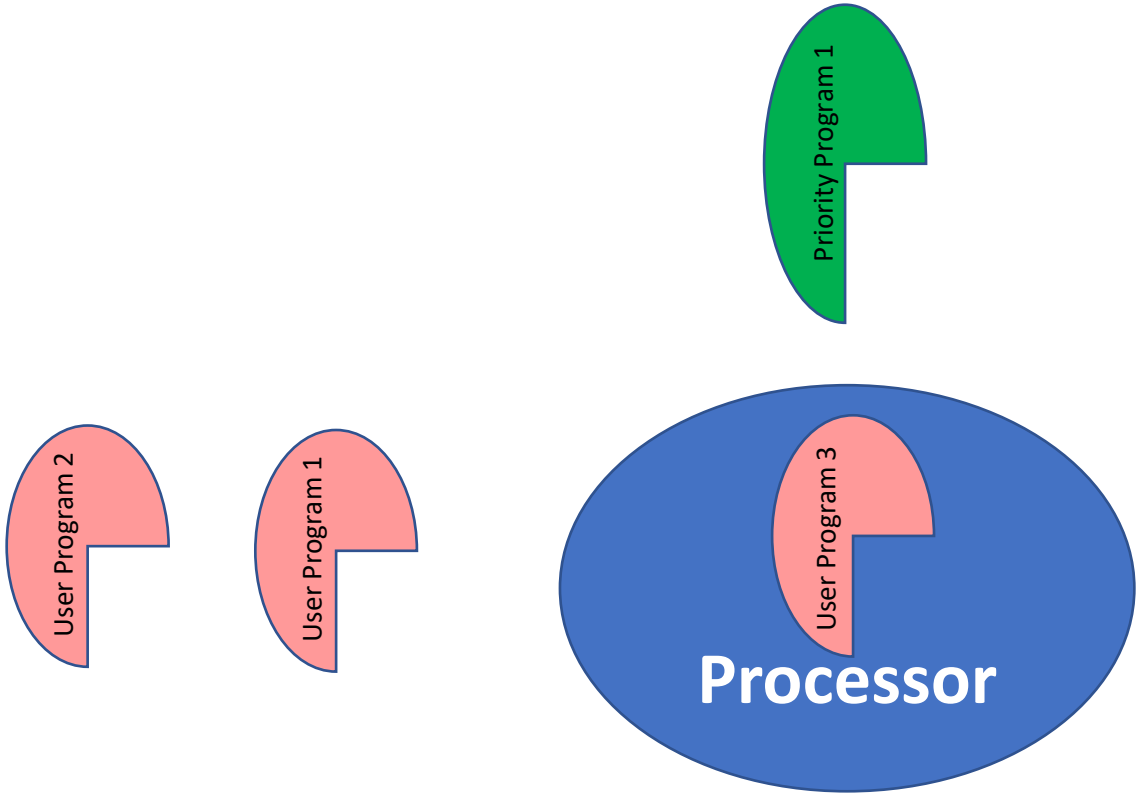
Context Switching between User Program 2 and User Program 3

User Program 3 is Executing

User Program 1 and User Program 2 are waiting for CPU time for execution

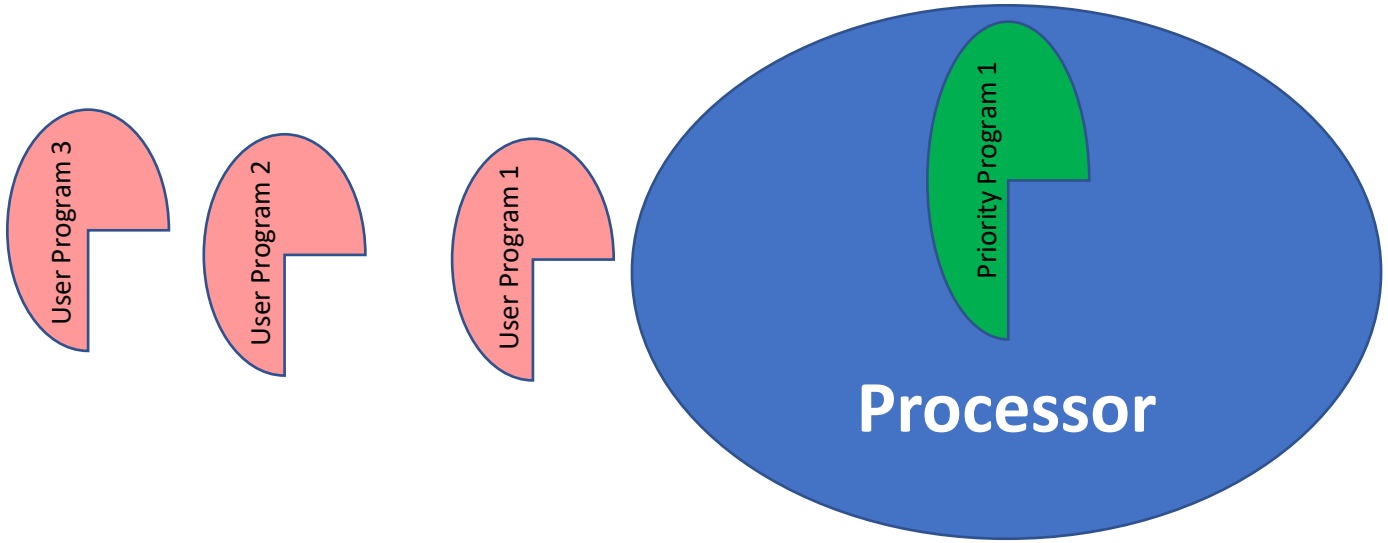
Microprocessor & Computer Architecture (µpCA)

Processor Modes



Priority Program Arrive when User Program 3 is executing.
User Program 1 and User Program 2 are waiting for CPU time for execution

Processor Modes

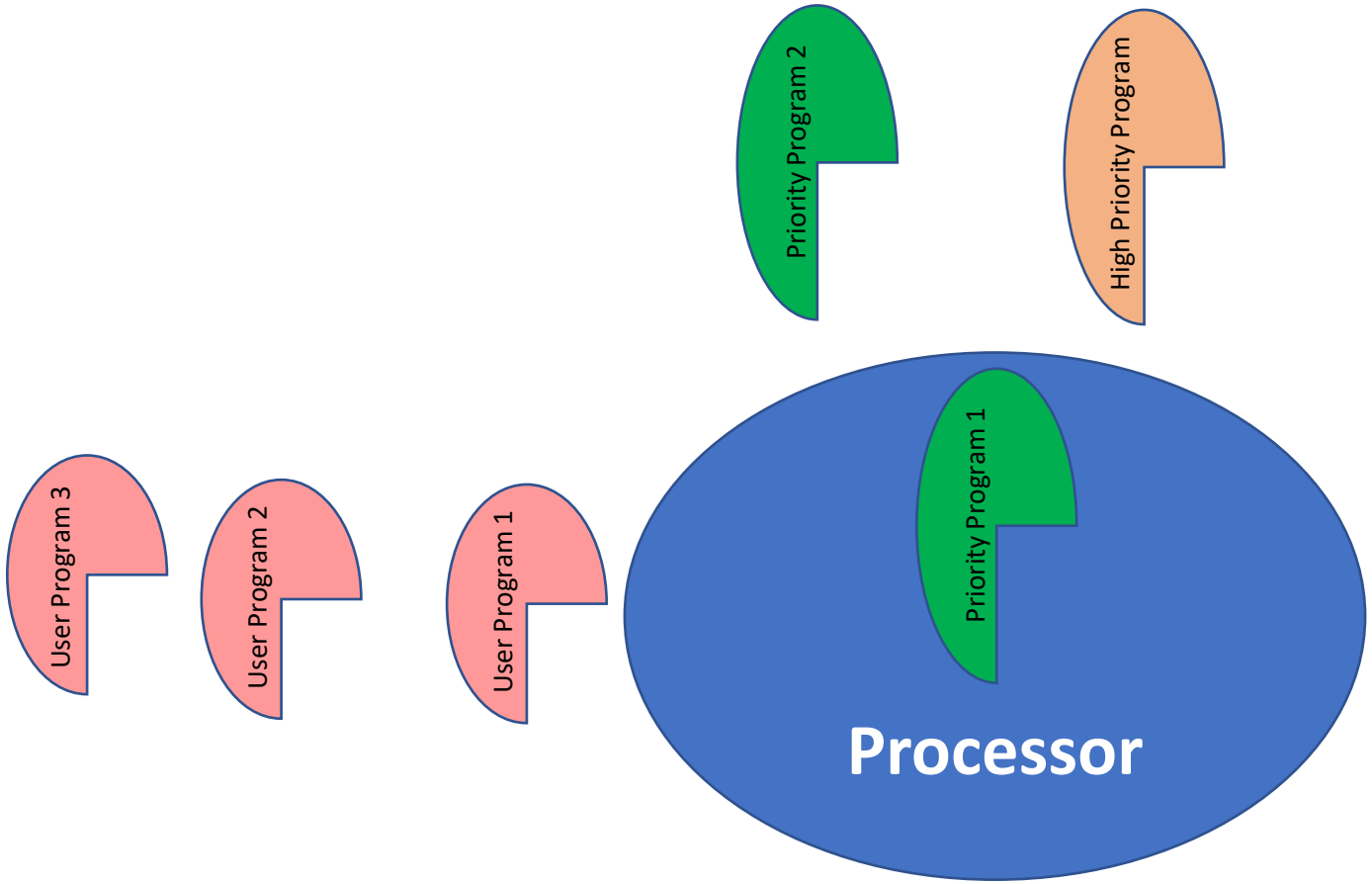


Context Switching between User Program 3 and Priority Program

Priority Program is Executing

User Program 1, 2 and 3 are waiting for CPU time for execution

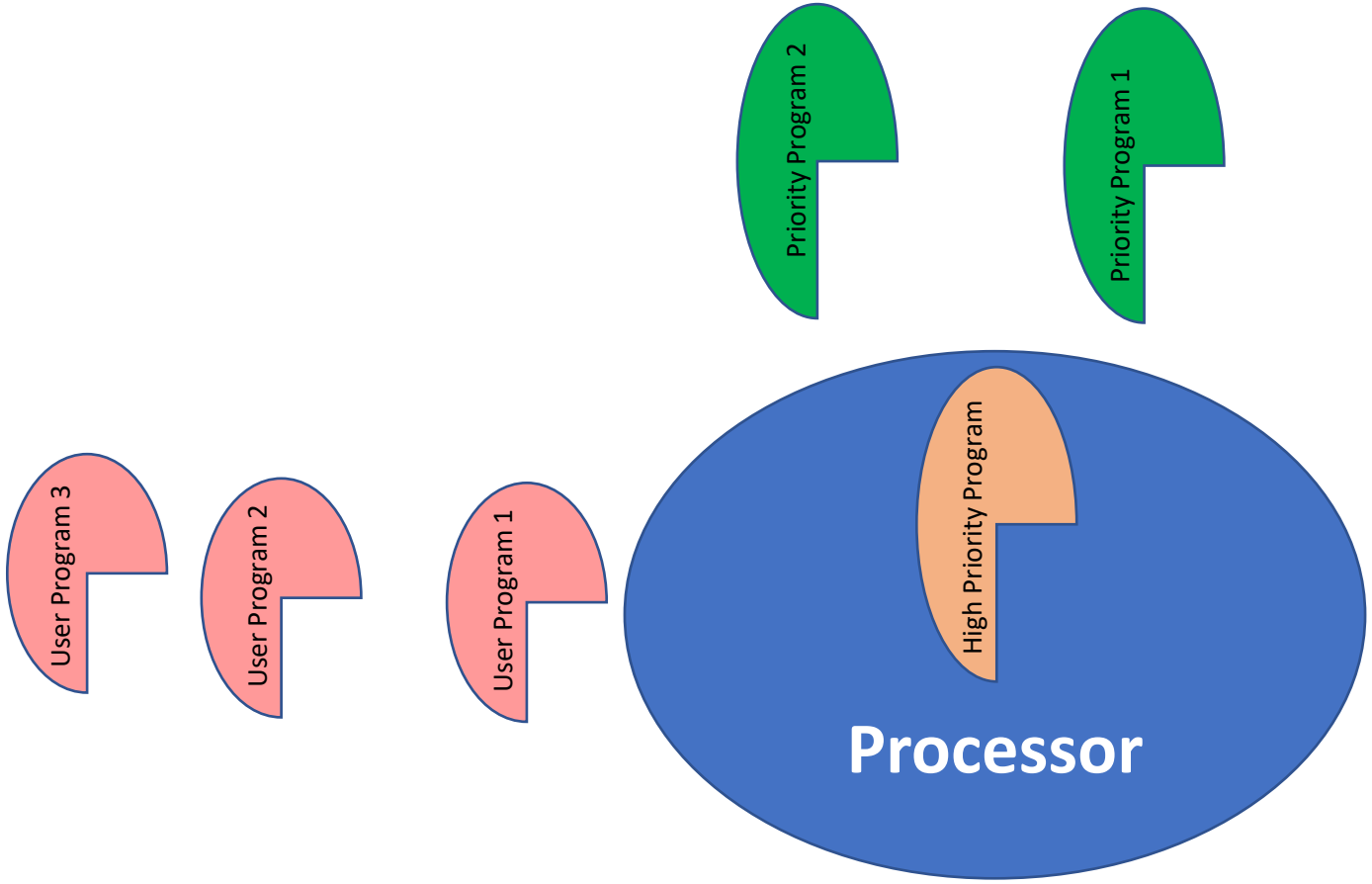
Processor Modes



High Priority Program and a Low Priority Program Arrive, when Priority program is Executing

User Program 1, 2 and 3 are waiting for CPU time for execution

Processor Modes



High Priority Program and a Low Priority Program Arrive, when Priority program is Executing
Context Switching between High Priority Program and Priority Program
High Priority Program is Executing
User Program 1, 2 and 3 are waiting for CPU time for execution

Processor Modes

The ARM has seven basic operating modes:

- **User** : Unprivileged mode under which most tasks run
- **FIQ** : Entered when a high priority (fast) interrupt is raised
- **IRQ** : Entered when a low priority (normal) interrupt is raised
- **Supervisor** : Entered on reset and when a Software Interrupt instruction is executed
- **Abort** : Used to handle memory access violations
- **Undef** : Used to handle undefined instructions
- **System** : Privileged mode using the same registers as user mode

Modes other than User mode are collectively known as Privileged modes. Privileged modes are used to service interrupts or exceptions, or to access protected resources.

Microprocessor & Computer Architecture (µpCA)

Next Class

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- Register Bank

THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Register Bank

Dr. D. C. Kiran
Department of Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Syllabus

Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- Register Bank

Unit 2: Pipelined Processor and Design

Unit 3: Memory Design

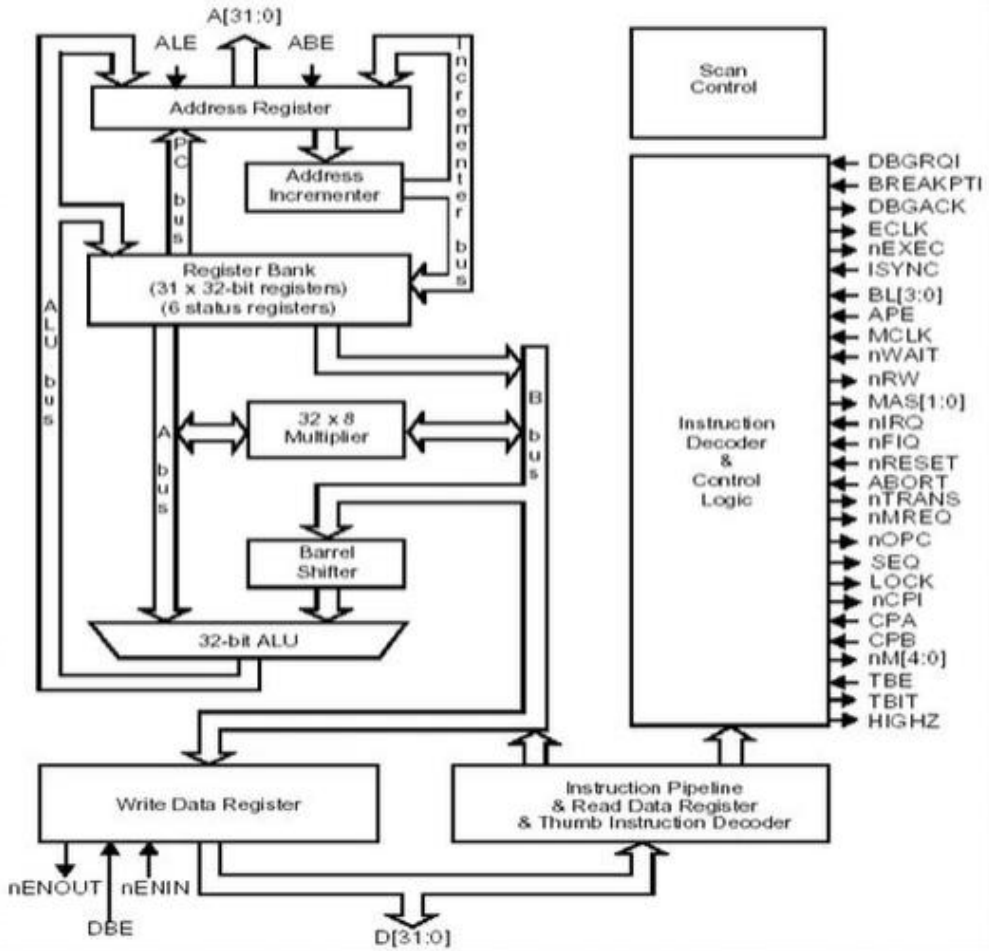
Unit 4: Input/Output Device Design

Unit 5: Advanced Architecture

ARM: [ACORN RISC MACHINE]

ARM7TDMI Processor

- Von Neumann Architecture
- 3-stage pipeline
  - fetch, decode, execute
- 32-bit Data Bus
- 32-bit Address Bus
- 37 32-bit registers
- 32-bit ARM instruction set
- 16-bit THUMB instruction set
- 32x8 Multiplier
- Barrel Shifter



Register Bank

- ARM has 37 registers all of which are 32-bits long.
  - 1 dedicated Program Counter (PC)
  - 1 dedicated Current Program Status Register (CPSR)
  - 5 dedicated Saved program Status Registers (SPSR)
  - 30 general purpose registers

- The current processor mode governs which of several banks is accessible. Each mode can access
  - a particular set of r0-r12 registers
  - a particular r13 (the stack pointer, SP) and r14 (the link register, LR)
  - the program counter, r15 (PC)
  - the current program status register, CPSR

Reference:
https://www.sciencedirect.com/topics/computer-science/banked-register

The ARM has Seven basic operating modes:

User & System

Common R0-R12, Dedicated R13 (SP), Dedicated R14(LR), Common R15 (PC) & CPSR

- **IRQ / Supervisor / Abort / Undef**

Common R0-R12, Dedicated R13 (SP), Dedicated R14(LR), Common R15 (PC) & CPSR and Dedicated SPSR

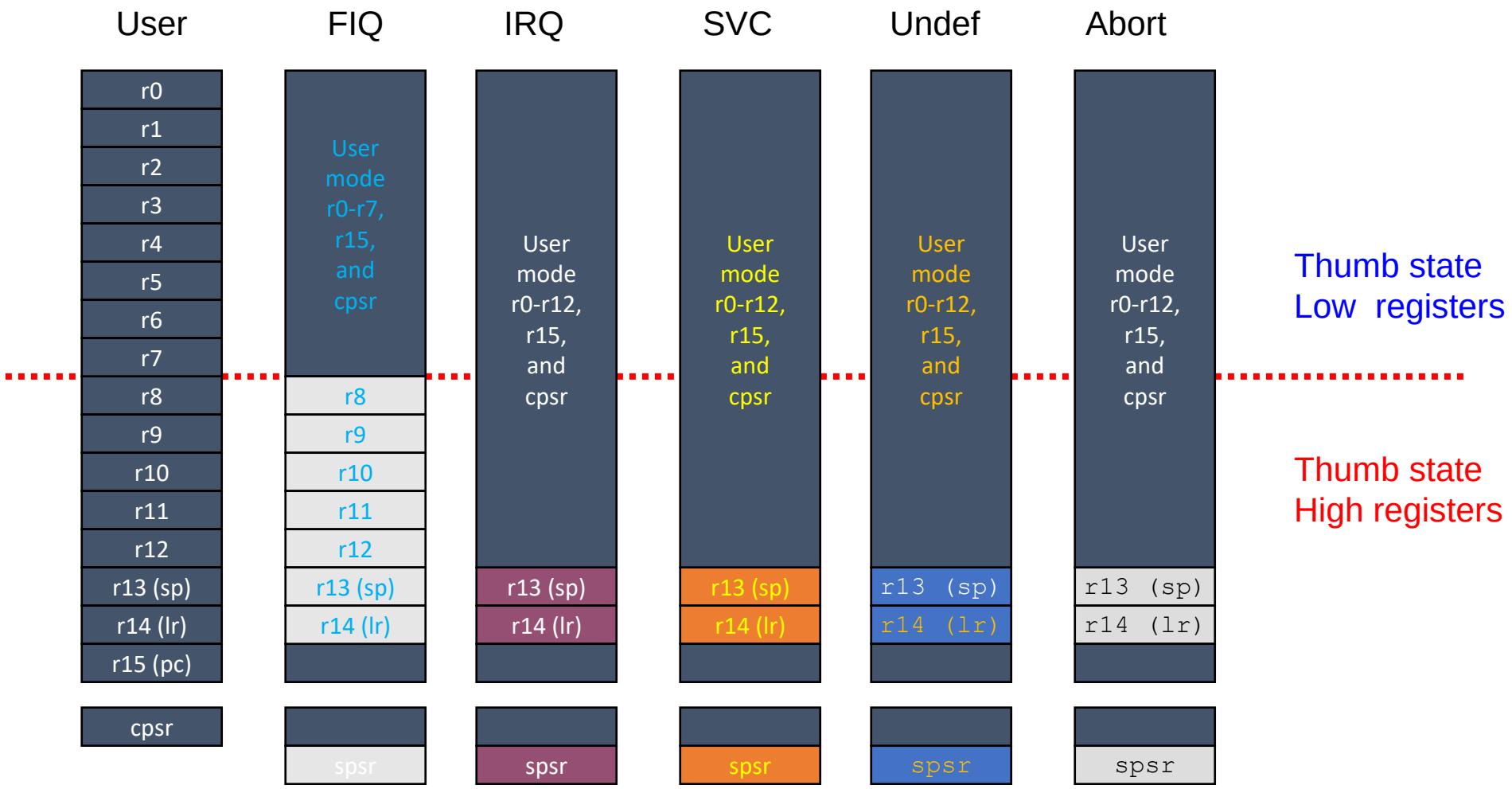
FIQ

Common R0-R7, Dedicated R8-R12, Dedicated R13 (SP), Dedicated R14(LR), Common R15 (PC) & CPSR and Dedicated SPSR

PC	1	Common
CPSR	1	Common
SPSR	5	All Priviliged mode
GPR	8	Common R0-R7
GPR	5	Common R8-R12
Spl GPR	5	FIQ R8-R12
SP	6	All Mode
LR	6	All Mode
	37	

Microprocessor & Computer Architecture (µpCA)

Register Organization Summary



Note: System mode uses the User mode register set

Status Register : CPSR & SPSR

Current Program Status Register

- ARM core uses CPSR to monitor & control internal operations.
- The current status of the program under execution, such as, result of current execution instruction is zero/ -ve are captured here.

Saved Program Status Register

- Processor, While Shifting one mode to another mode, CPSR will be copied to SPSR.
- SPSR will be copied back to CPSR when it return back to previous mode.

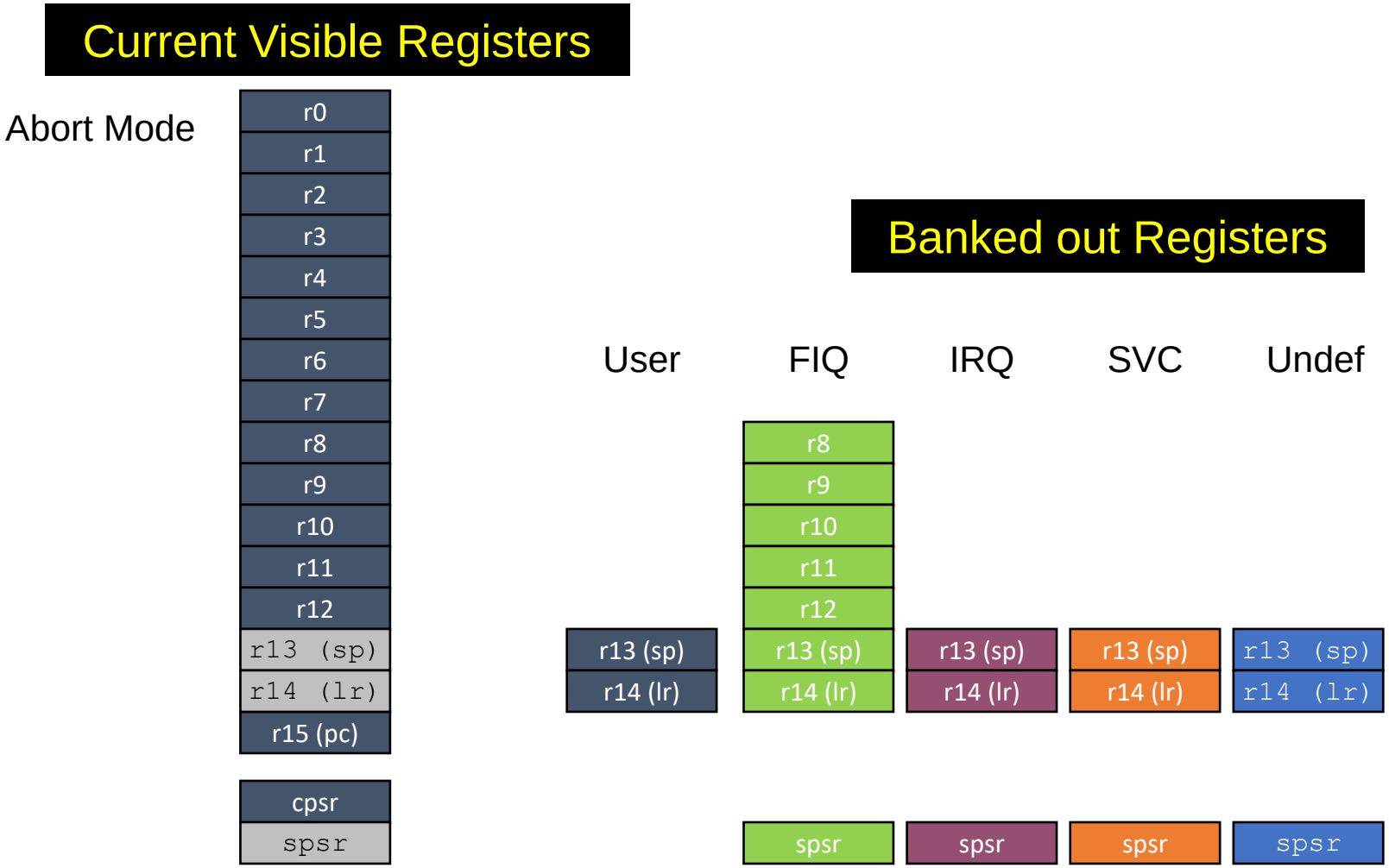
Example:

Suppose Processor is in IRQ mode of operation and if FIQ request arrives then processor has to switch itself to FIQ mode of operation. First CPSR get copied to SPSR of IRQ mode, then processor will serve FIQ mode. After serving FIQ mode processor should return to IRQ mode and should resume its working. So SPSR of IRQ mode gets copied again into CPSR to serve in the IRQ mode.

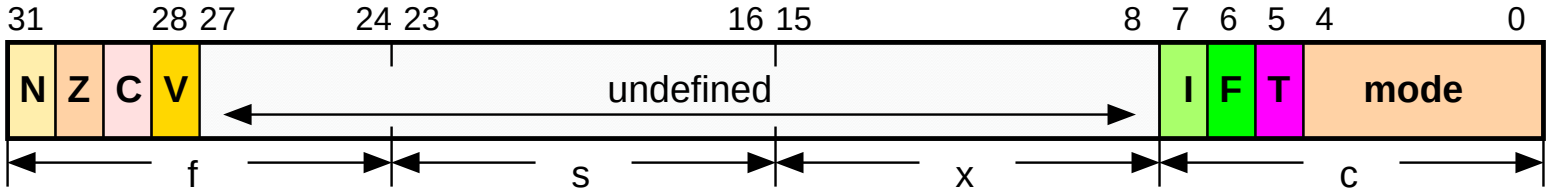
Click here for Reference

Microprocessor & Computer Architecture (µpCA)

The ARM Register Set

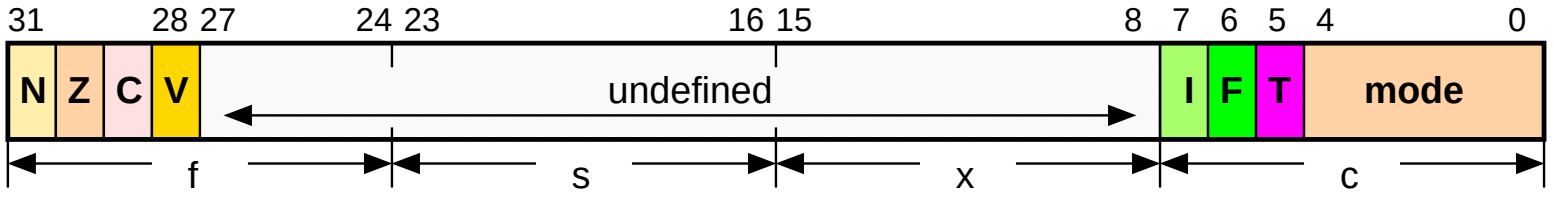


Status Register (SR)



SR fields are divided into four fields, each 8-bit wide:

- Flags (f)
- Status (s) : *Reserved for Future*
- Extension (x) : *Reserved for Future*
- Control (c).



- **N: Negative**: Set to 1 if the Result of previous instruction is Negative.
- **Z: Zero**: Set to 1 if the Result of previous instruction is Zero.
- **C: Carry**: Set to 1, if result of either an Arithmetic or Shifter produce a carry-out,
- **V: oVerflow**: Set to 1, if previous instruction produce an overflow into the sign bit.

- **Mode bits**

10000	**User**
10001	**FIQ**
10010	**IRQ**
10011	**Supervisor**
10111	**Abort**
11011	**Undefined**
11111	**System**

Interrupt Disable bits.

I = 1: Disables the IRQ.

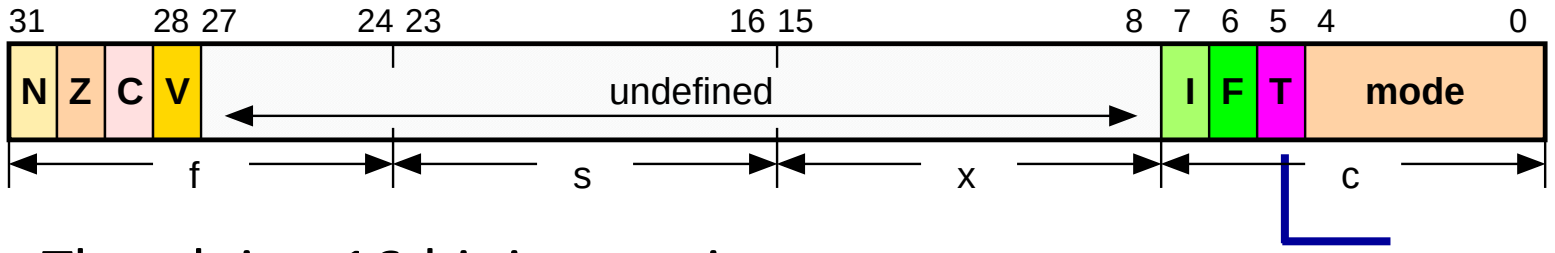
F = 1: Disables the FIQ.

T Bit (Arch. with Thumb mode only)

T = 0: Processor in ARM state

T = 1: Processor in Thumb state

Thumb Mode



- Thumb is a 16-bit instruction set
  - Optimized for code density from C code (~65% of ARM code size)
  - Improved performance from memory with a narrow data bus
  - Subset of the functionality of the ARM instruction set
  - Only Low Registers R0-R7 are used
  - Constants are of limited size.
  - Inline barrel shifter not used.

I & F Bits

I	F	
1	1	FIQ is served, IRQ and FIQ is disabled
1	0	IRQ is served, IRQ is disabled & FIQ is enabled
0	1	**FIQ is served, IRQ is enabled & FIQ is disabled (Not Allowed)**
0	0	USER program is served, IRQ and FIQ both are enabled

Microprocessor & Computer Architecture (µpCA)

Next Class

PES UNIVERSITY ONLINE

Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ARM Program structure
- ARM Instruction Format

THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

PES UNIVERSITY ONLINE

Microprocessor & Computer Architecture (µpCA)

ARM Program Structure & Instruction Format

Dr. D. C. Kiran
Department of Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Syllabus

Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ARM Program structure
- ARM Instruction Format

Microprocessor & Computer Architecture (µpCA)

ARM Program Structure

Address	Instruction & Data
	.text
Address of Instruction 1	ARM Instruction_1
Address of Instruction 2	ARM Instruction_3
Address of Instruction 3	ARM Instruction_3
…….	…….
…….	…….
Address of Instruction n	ARM Instruction_n
	.data
Address of Data1	Declaration of variable 1
Address of Data2	Declaration of variable 2
…….	…….
…….	…….
Address of Data n	Declaration of variable n

Note: Blue color depict the code written by the programmer
Red color depict the address assigned during execution

Microprocessor & Computer Architecture (µpCA)

ARM Program Structure

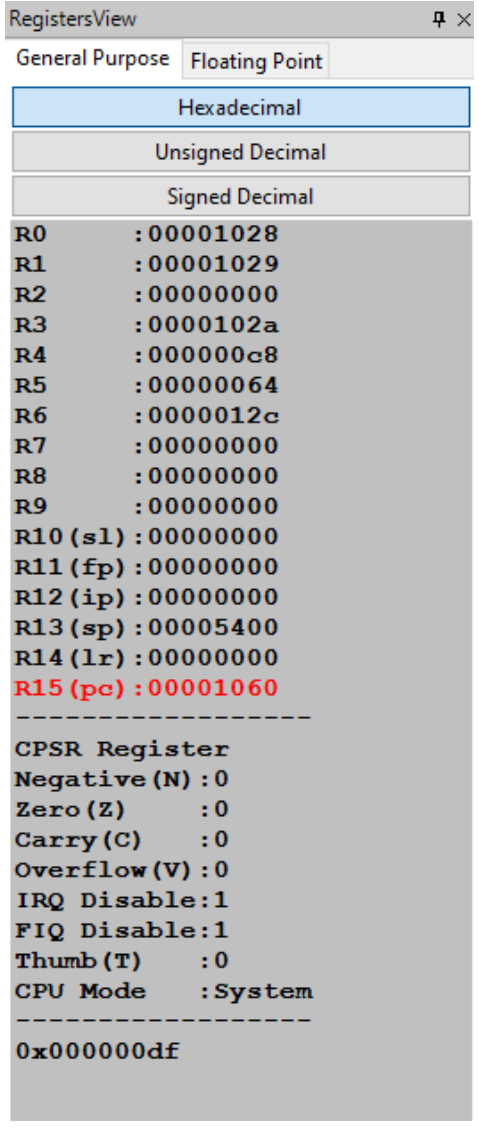
Address	Instruction	Meaning
	.text	
00001000:EF9F0014	LDR R0, =a	Load the Address of a to R0
00001004:EF9F1014	LDR R1,=b	Load the Address of b to R1
00001008:EF9F3014	LDR R3, =c	Load the Address of c to R3
0000100C:E5D14000	LDR R4, [r1]	Load the value (100) to R4
00001010:E5D05000	LDR R5, [r0]	Load the value (200) to R5
00001014:E0846005	Add R6, R4, R5	Add R4 & R5
00001018:E00360B0	STR R6, [r3]	Store the result(300) in the address specified in R3
	.data	
00001028:	a: .word 100	Variable ***a*** of data type ***word***
00001029:	b: word 200	Variable ***b*** of data type ***word***
0000102A:	c: word 0	Variable ***c*** of data type ***word***

Note: the datatype may be byte, half word, asciz

Microprocessor & Computer Architecture (µpCA)

Register View (ARMSIM Simulator)

```
                        .text
00001000:EF9F0014       LDR R0, =a
00001004:EF9F1014       LDR R1,=b
00001008:EF9F3014       LDR R3, =c
0000100C:E5D14000       LDR R4, [r1]
00001010:E5D05000       LDR R5, [r0]
00001014:E0846005       ADD R6, R4, R5
00001018:E00360B0       STR R6, [r3]

                        .data
00001028:               a: .word 100
00001029:               b: word 200
0000102A:               c: word 0
```



Salient features of ARM Instructions

- Each Instruction is 32 bit wide
- Load-store architecture
- 3-address instructions.
- Conditional execution of every instruction.
- Possible to load/store multiple register at once.
- Possible to combine shift and ALU operations in a single instruction.
- No Memory-Memory Operations

General Format of Instruction

```
MNEMONIC{condition}{S} {Rd},Operand1,Operand2
```

MNEMONIC - Short name of the instruction. ***Eg: ADD,SUB….***

{condition} - Condition that is needed to be met in order for the instruction to be executed ***EG: EQ, MI,GT,LT,LE,AL,NE***

{S} - An optional suffix. If S is specified, the condition flags are updated on the result of the operation . ***Eg: To set N,O, C, V of CPSR***

{Rd} - Register (destination) for storing the result of the instruction

Operand1 - First operand. Either a register or an immediate value

Operand2 - Second (flexible) operand. Can be an immediate value (number) or a register with an optional shift

General Format of Instruction Example

Example 1:

MOV r0, #-10
MOV r1, #-10
ADD r2, r1,r0

```
R0      :-10
R1      :-10
R2      :-20
R3      :0
R4      :0
R5      :0
R6      :0
R7      :0
R8      :0
R9      :0
R10(sl):0
R11(fp):0
R12(ip):0
R13(sp):21504
R14(lr):0
R15(pc):70656
------------------
CPSR Register
Negative(N):0
Zero(Z)     :0
Carry(C)    :0
Overflow(V):0
IRQ Disable:1
FIQ Disable:1
Thumb(T)    :0
CPU Mode    :System
------------------
```

General Format of Instruction Example

Example2:

MOV r0, #-10
MOV r1, #-10
ADDS r2, r1,r0

```
R0      :-10
R1      :-10
R2      :-20
R3      :0
R4      :0
R5      :0
R6      :0
R7      :0
R8      :-10
R9      :0
R10(sl):0
R11(fp):0
R12(ip):0
R13(sp):21504
R14(lr):0
R15(pc):70656
------------------
CPSR Register
Negative(N):1
Zero(Z)    :0
Carry(C)   :1
Overflow(V):0
IRQ Disable:1
FIQ Disable:1
Thumb(T)   :0
CPU Mode   :System
------------------
```

General Format of Instruction Example

Example 3

```
.text
MOV r0,#-44
MOV r1,#4
 SUBS r2,r0,r1
 ADDMIS r0,r0,r1
.end
```

- Subtract r1 from r0
- Add only if the result of SUB is -ve

(Here result of SUB is –ve so add is executed

```
R0      :-40
R1      :4
R2      :-48
R3      :0
R4      :0
R5      :0
R6      :0
R7      :0
R8      :0
R9      :0
R10(sl):0
R11(fp):0
R12(ip):0
R13(sp):21504
R14(lr):0
R15(pc):70656
------------------
CPSR Register
Negative(N):1
Zero(Z)     :0
Carry(C)    :0
Overflow(V):0
IRQ Disable:1
FIQ Disable:1
Thumb(T)    :0
CPU Mode    :System
------------------
```

General Format of Instruction Example

Example 4

```
.text
MOV r0,#44
MOV r1,#-4
 SUBS r2,r0,r1
 ADDMIS r0,r0,r1
.end
```

- Subtract r1 from r0
- Add only if the result of SUB is -ve

(Here result of SUB is not –ve so add is not executed

```
R0      :44
R1      :-4
R2      :48
R3      :0
R4      :0
R5      :0
R6      :0
R7      :0
R8      :0
R9      :0
R10(sl):0
R11(fp):0
R12(ip):0
R13(sp):21504
R14(lr):0
R15(pc):70656
------------------
CPSR Register
Negative(N):0
Zero(Z)    :0
Carry(C)   :0
Overflow(V):0
IRQ Disable:1
FIQ Disable:1
Thumb(T)   :0
CPU Mode   :System
------------------
```

General Format of Instruction Example

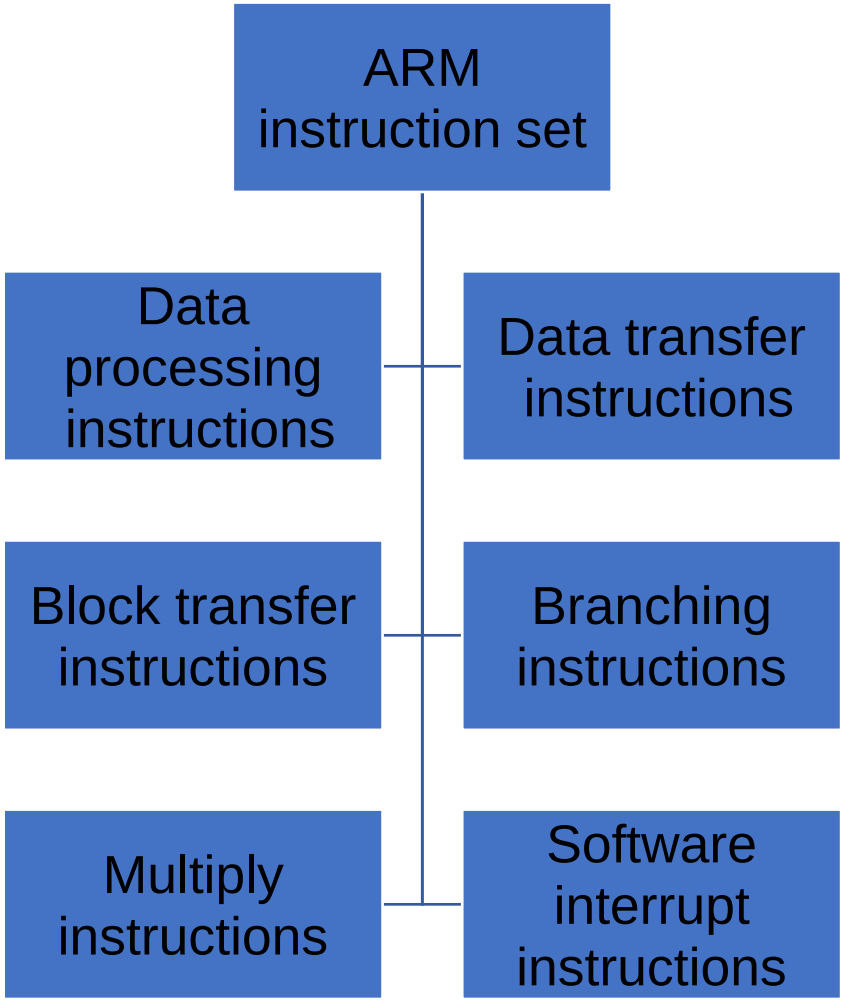
Example 5

```
.text
MOV r0,#44
MOV r1,#4
CMP r1,r0
ADDEQ r0,r0,r1
.end
```

- Compare r1 and r0
- Add only of the comparison is true i.e N=0
  (Here comparison fails i.e N=1)

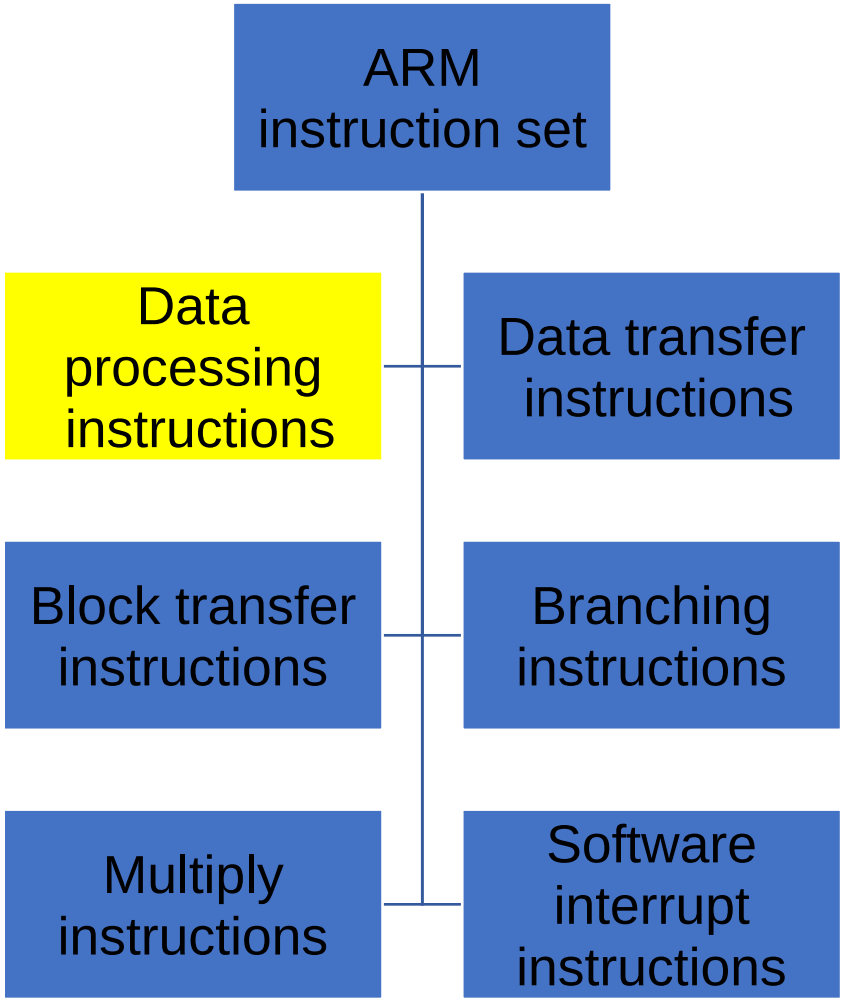
```
R0      :44
R1      :4
R2      :0
R3      :0
R4      :0
R5      :0
R6      :0
R7      :0
R8      :0
R9      :0
R10(sl):0
R11(fp):0
R12(ip):0
R13(sp):21504
R14(lr):0
R15(pc):70656
------------------
CPSR Register
Negative(N):1
Zero(Z)    :0
Carry(C)   :0
Overflow(V):0
IRQ Disable:1
FIQ Disable:1
Thumb(T)   :0
CPU Mode   :System
------------------
```

ARM Instruction Set



Microprocessor & Computer Architecture (µpCA)

Next Session



Click Here for reference

THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

PES UNIVERSITY ONLINE

Microprocessor & Computer Architecture (µpCA)

Data Processing Instructions

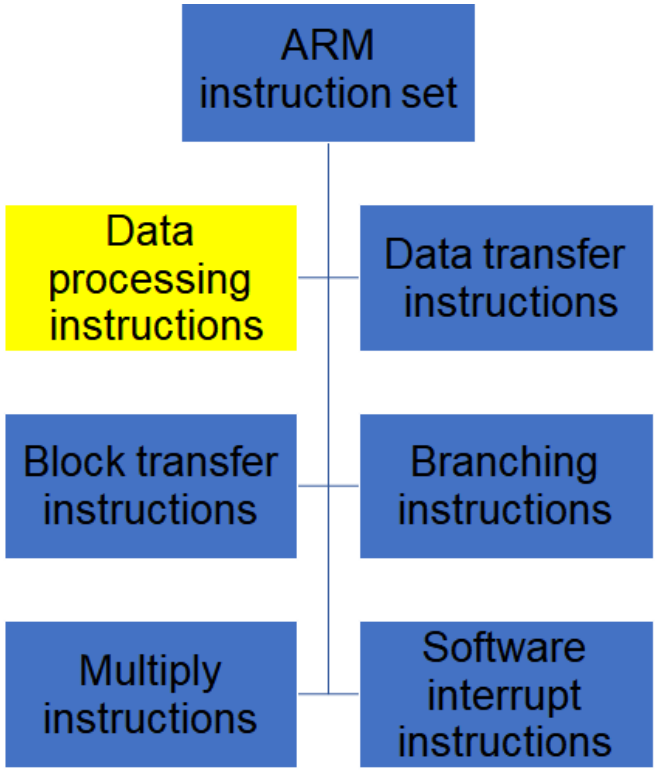
Dr. D. C. Kiran
Department of Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

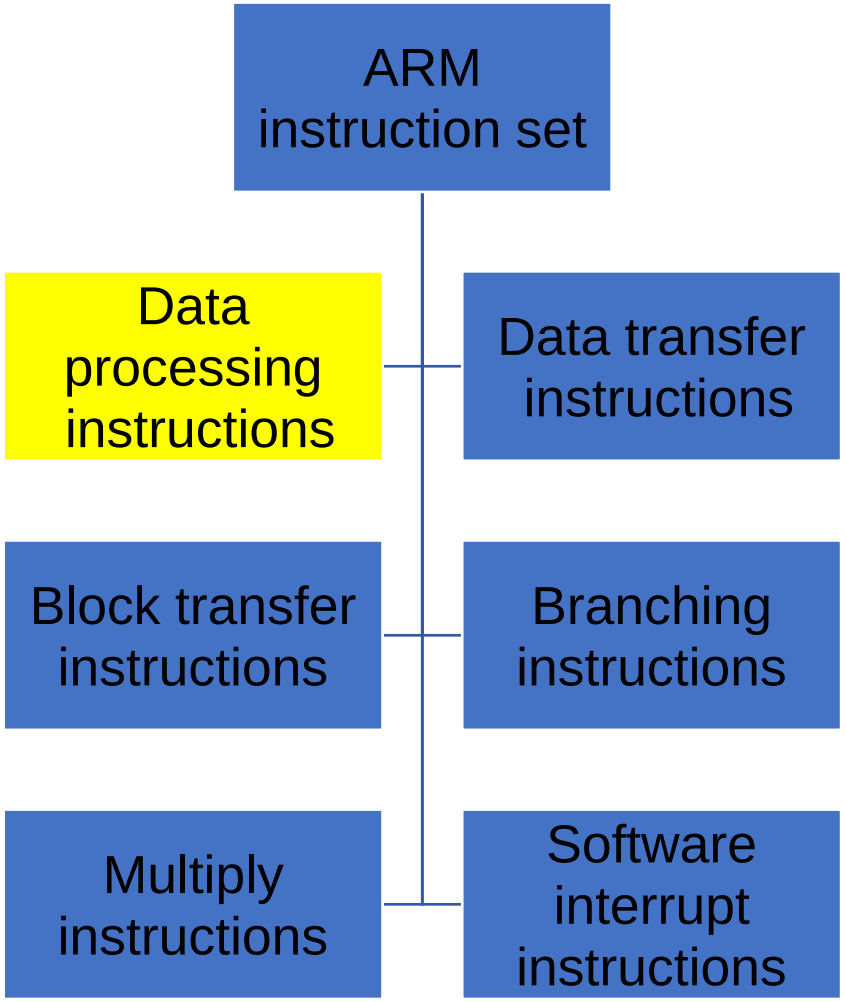
Syllabus

Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ~~ARM Program structure~~
- ~~ARM Instruction Format~~
- **ARM INSTRUCTION SET**
  - Data Processing Instructions
  - Data Movement Instruction
  - Arithmetic Instruction
  - Multiword Arithmetic

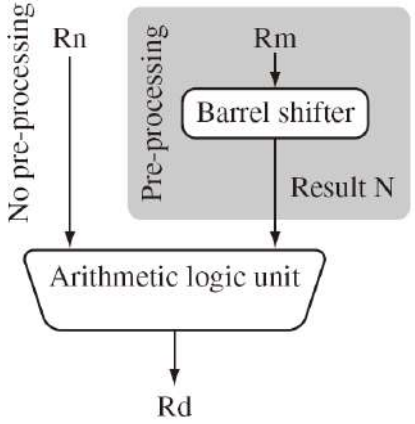


ARM INSTRUCTION SET



Data processing instructions

- Largest family of ARM instructions, all sharing the same instruction format.
- Contains:
  - Arithmetic operations
  - Comparisons (no results - just set condition codes)
  - Logical operations
  - Data movement between registers



- Remember, this is a load / store architecture
  - These instruction only work on registers, ***NOT*** memory.
- They each perform a specific operation on one or two operands.
  - First operand always a register - Rn
  - Second operand sent to the ALU via barrel shifter.

Data Movement

- Operations are:
  - MOV operand2
  - MVN NOT operand2

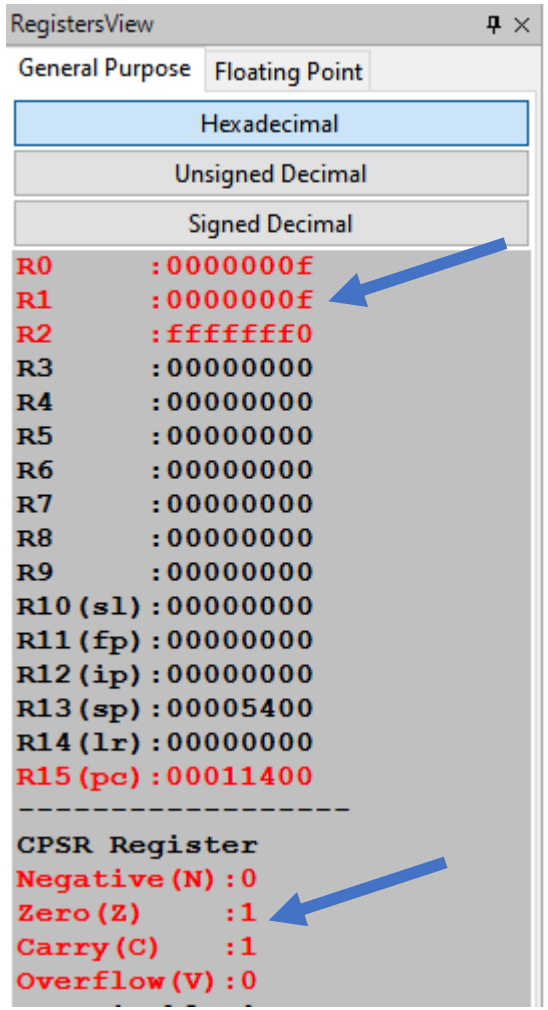
  Note that these make no use of operand1.
- Syntax:
  - <Operation>{<cond>}{S} Rd, Operand2
- Examples:
  - MOV r0, r1
  - MOVS r2, #10
  - MVNEQ r1,#0

Microprocessor & Computer Architecture (µpCA)

```
.text
MOVS R0, #-256
.end
```

```
R0      :-256
R1      :0
R2      :0
R3      :0
R4      :0
R5      :0
R6      :0
R7      :0
R8      :0
R9      :0
R10(sl):0
R11(fp):0
R12(ip):0
R13(sp):21504
R14(lr):0
R15(pc):70656
------------------
CPSR Register
Negative(N):1
Zero(Z)     :0
Carry(C)    :0
```

```
.text
MOV R0,#5
MOV R1,#5
CMP R0,R1
MVNEQ R2, #25
.end
```

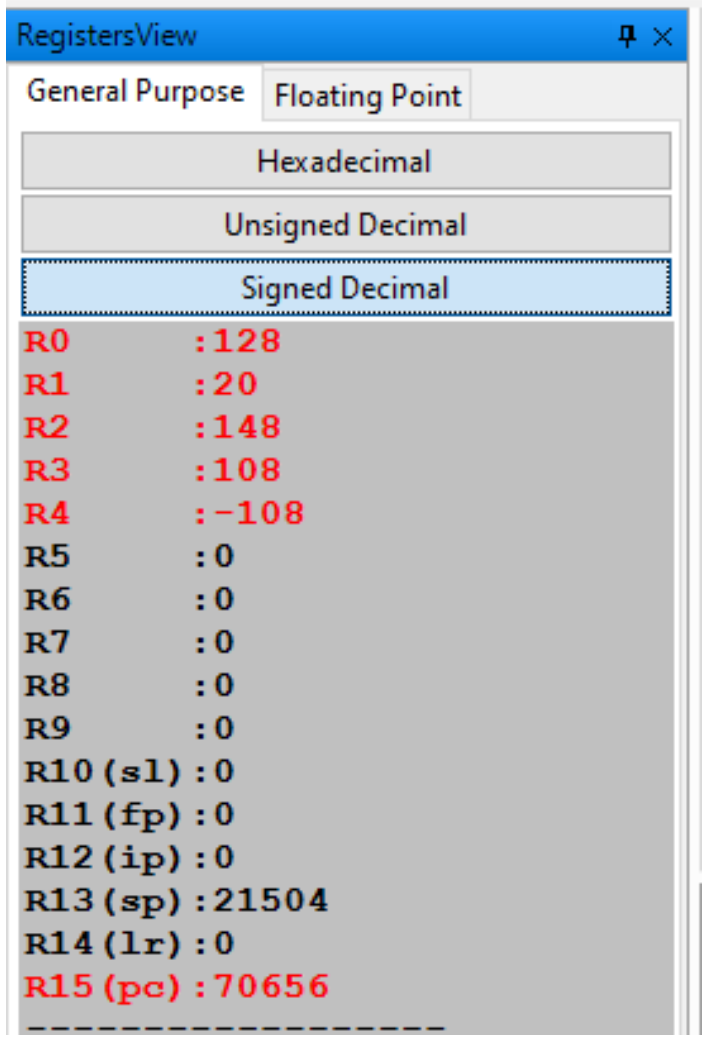


Arithmetic Operations

- Operations are:
  - ADD operand1 + operand2
  - SUB operand1 - operand2
  - RSB operand2 - operand1
- Syntax:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- Examples
  - ADD r0, r1, r2
  - SUBGT r3, r3, #1
  - RSBLES r4, r5, #5

Example: Data Processing Instructions

```
;ADD 2 numbers loaded from register
.text
MOV r0, #0x80
MOV r1, #20
ADD r2, r0, r1

;Sub 2 numbers loaded from register
SUB r4, r1, r0
SUB r3, r0, r1
.end
```



Arithmetic Operations With Carry

- **ADC:** Operand1 + Operand 2 + CPSR.c
- **SBC:** Operand 1 – Operand2 –NOT(CPSR.c)

  or

  Operand1 - Operand2 + carry -1
- **RSC:** Operand2-Operand 1 - NOT(CPSR.c)

  or

  Operand2 – Operand1 + carry -1

NOTE: For SBC & RSC

Since Subtraction is performed using adder, the Operand2 & Carry in is inverted and fed to adder or Invert the Operand 2 first and subtract carry later.

- Operand1+(-Operand2)
- Operand1+~Operand2+1

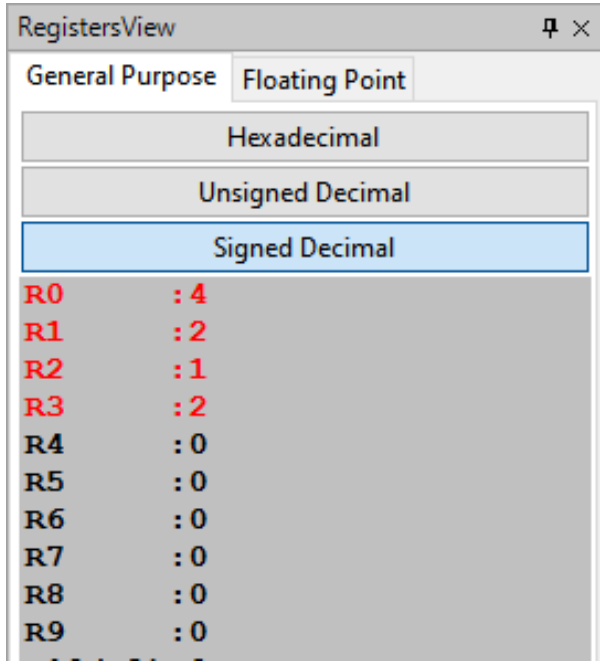
Arithmetic Operations With Carry

SBC: Operand1 - Operand2 + carry -1

or

Operand 1 – Operand2 –NOT(CPSR.c)

```
.text
MOV r0, #4
MOV r1, #2
SUB r3,r0,r1
SBC r2, r0, r1
.end
```



~~Wrong application of SBC~~

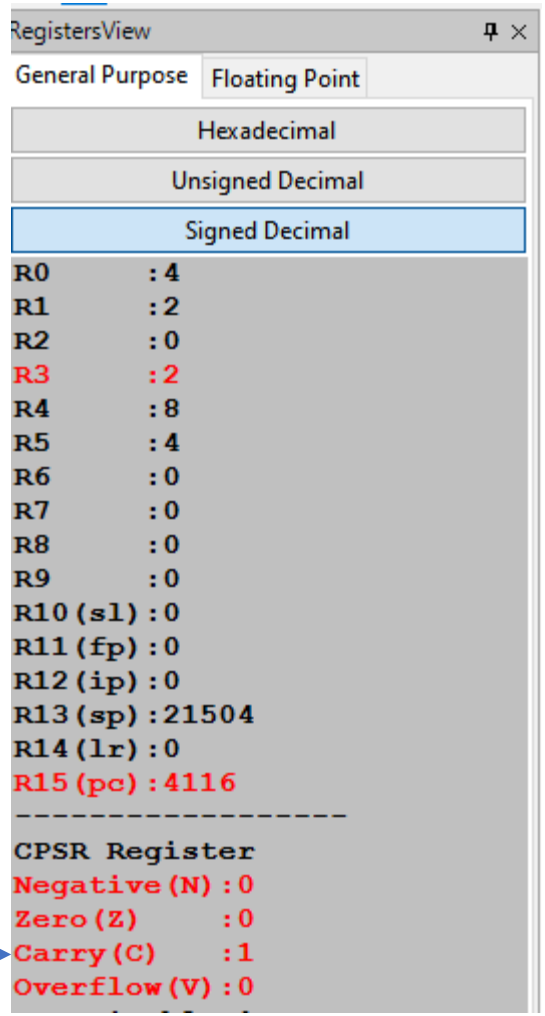
Microprocessor & Computer Architecture (µpCA)

Arithmetic Operations With Carry

SBC: operand1 - operand2 + carry -1

```
.text    MOV r0, #4
         MOV r1, #2
         MOV r4, #8
         MOV r5, #4
         SUBS r3,r0,r1
         SBC r2, r4, r5
.end
```

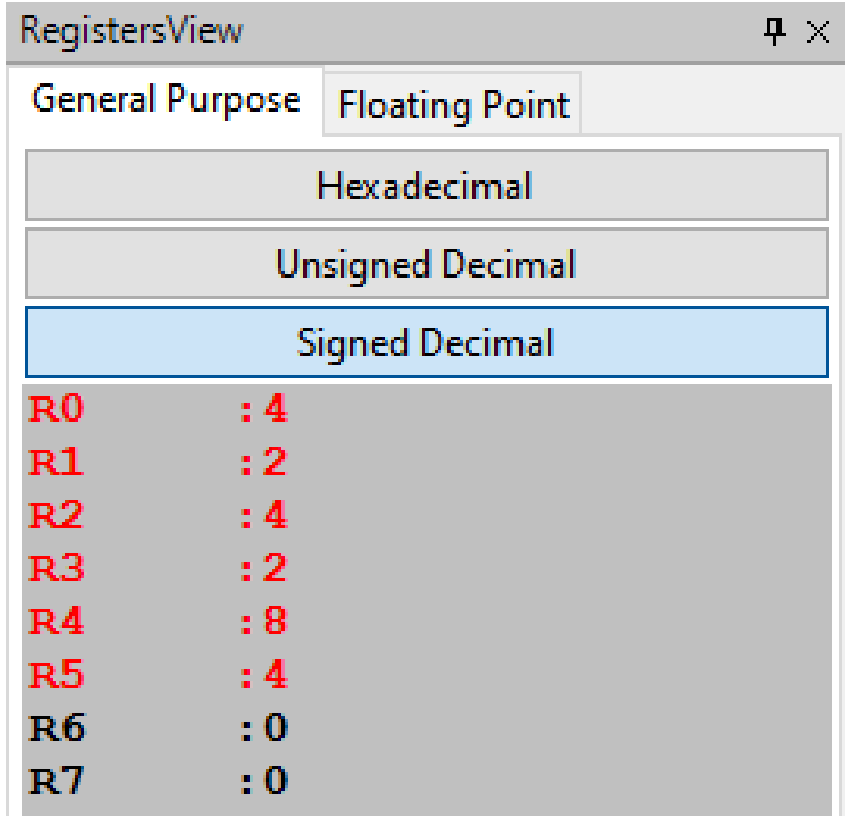
After executing SUBS



Microprocessor & Computer Architecture (µpCA)

Arithmetic Operations With Carry

```
.text   MOV r0, #4
        MOV r1, #2
        MOV r4, #8
        MOV r5, #4
        SUBS r3,r0,r1
        SBC r2, r4, r5
.end
```



Microprocessor & Computer Architecture (µpCA)

Multiword Arithmetic

If more than 32 bit or 1 word operations need to be performed, ***ADC, SBC,RSC*** instructions are used.

- **Example 1:** 64 Bit Integer → | **32 Bit MSB Word** | **32 Bit LSB Word** |
- Let ***A*** be a 64 bit data. ***r0*** will have least significant word and ***r1*** will have most significant word.
- Let ***B*** be a 64 bit data. ***r2*** will have least significant word and ***r3*** will have most significant word.
- Result of adding ***A*** and ***B*** will be in **r4** & **r5**.

```
ADDS r4,r0,r2 ; adding the least significant words
ADC r5,r1,r3 ; adding the most significant words
```

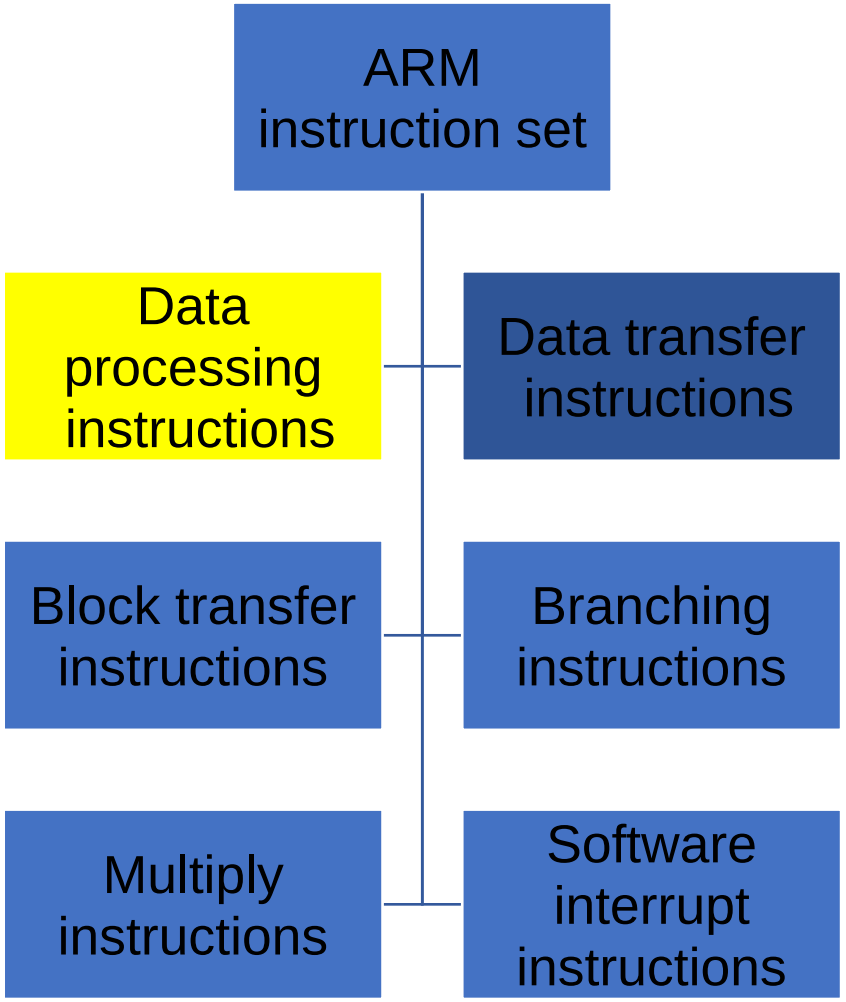
Example 2: These instructions subtract one 96-bit integer from another:

```
SUBS r3,r6,r9
SBCS r4,r7,r10
SBC r5,r8,r11
```

32 Bit MSB Word	**32 Bit Word**	**32 Bit LSB Word**

96 Bit Integer

Next Class: Barrel Shifter



THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

PES UNIVERSITY ONLINE

Microprocessor & Computer Architecture (µpCA)

Data Processing Instructions

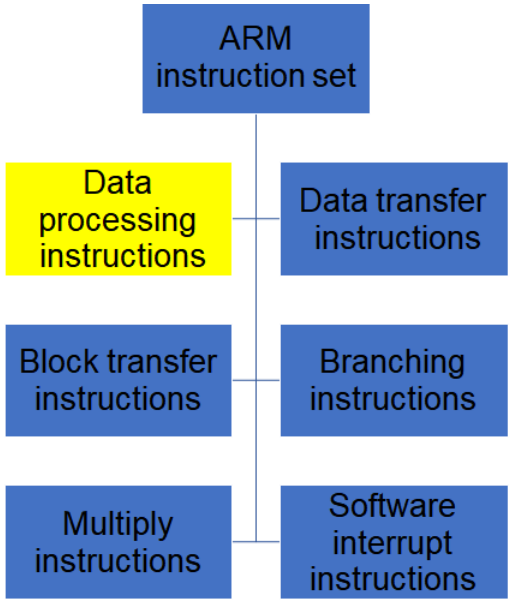
Dr. D. C. Kiran
Department of Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Syllabus

Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ~~ARM Program structure~~
- ~~ARM Instruction Format~~
- **ARM INSTRUCTION SET**
  - **Data Processing Instructions**
  - ~~Data Movement Instruction~~
  - ~~Arithmetic Instruction~~
  - ~~Multiword Arithmetic~~
  - ~~Barrel Shifter~~
  - Logical and Comparison Instruction



Comparison Instruction

Syntax:

<Operation>{<cond>} Rn, Operand2

- `CMP  R1, R2      @ set cc on R1-R2`
- `CMN  R1, R2      @ set cc on R1+R2`
- `TST  R1, R2      @ set cc on R1 and R2`
- `TEQ  R1, R2      @ set cc on R1 xor R2`

```
.text
        MOV R0, #25
        MOV R1, #256
        CMP R0,R1
.end
```

```
R0      :25
R1      :256
R2      :0
R3      :0
R4      :0
R5      :0
R6      :0
R7      :0
R8      :0
R9      :0
R10(sl):0
R11(fp):0
R12(ip):0
R13(sp):21504
R14(lr):0
R15(pc):70656
------------------
CPSR Register
Negative(N):1
Zero(Z)     :0
Carry(C)    :0
Overflow(V):0
```

```
.text
        MOV R0, #256
        MOV R1, #25
        CMP R0,R1
.end
```

```
R0      :256
R1      :25
R2      :0
R3      :0
R4      :0
R5      :0
R6      :0
R7      :0
R8      :217
R9      :0
R10(sl):0
R11(fp):0
R12(ip):0
R13(sp):21504
R14(lr):0
R15(pc):70656
------------------
CPSR Register
Negative(N):0
Zero(Z)     :0
Carry(C)    :1
Overflow(V):0
```

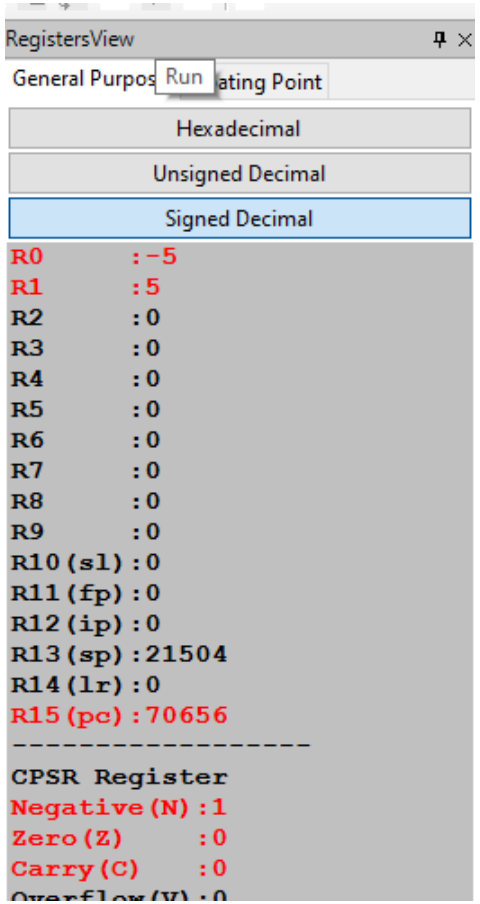
```
.text
        MOV R0, #256
        MOV R1, #256
        CMP R0,R1
.end
```

```
R0      :256
R1      :256
R2      :0
R3      :0
R4      :0
R5      :0
R6      :0
R7      :0
R8      :0
R9      :0
R10(sl):0
R11(fp):0
R12(ip):0
R13(sp):21504
R14(lr):0
R15(pc):70656
------------------
CPSR Register
Negative(N):0
Zero(Z)     :1
Carry(C)    :1
Overflow(V):0
```

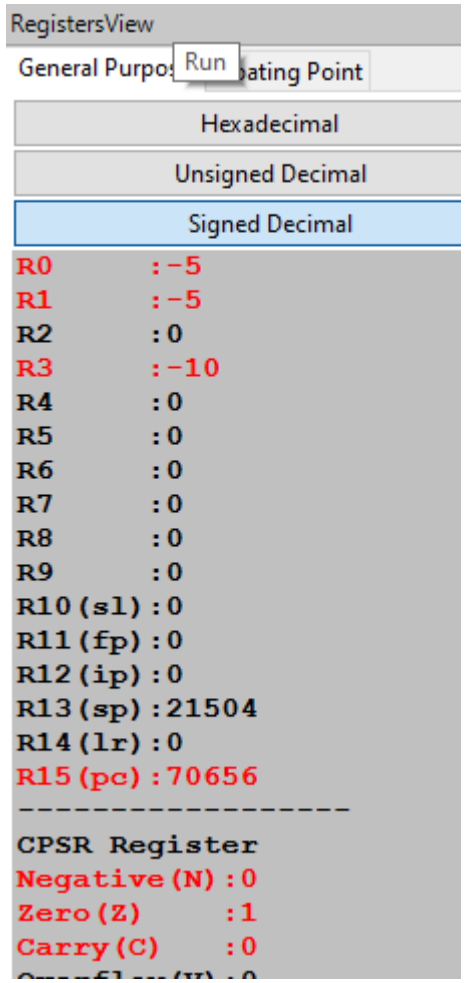
Microprocessor & Computer Architecture (µpCA)

Test (TST) & Test Equivalence (TEQ)

```
.text
MOV R0,#-5
MOV R1,#5
TEQ R0,R1
ADDEQ R3,R0,R1
.end
```




```
.text
MOV R0,#-5
MOV R1,#-5
TEQ R0,R1
ADDEQ R3,R0,R1
.end
```



Microprocessor & Computer Architecture (µpCA)

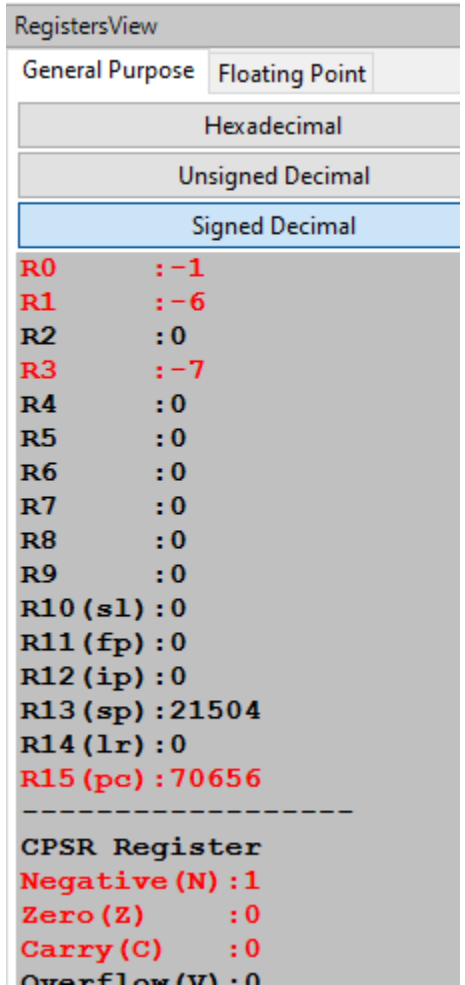
Test (TST) & Test Equivalence (TEQ)

TST perform AND operation against Operand 1 and Operand 2
Update the flags of CPSR based on the register
Used to check if any flag is set.

Example1 :

```
.text
MOV R0,#-1
MOV R1,#-6
TST R0,R1
ADDMI R3,R0,R1
.end
```

1111
1010
1010

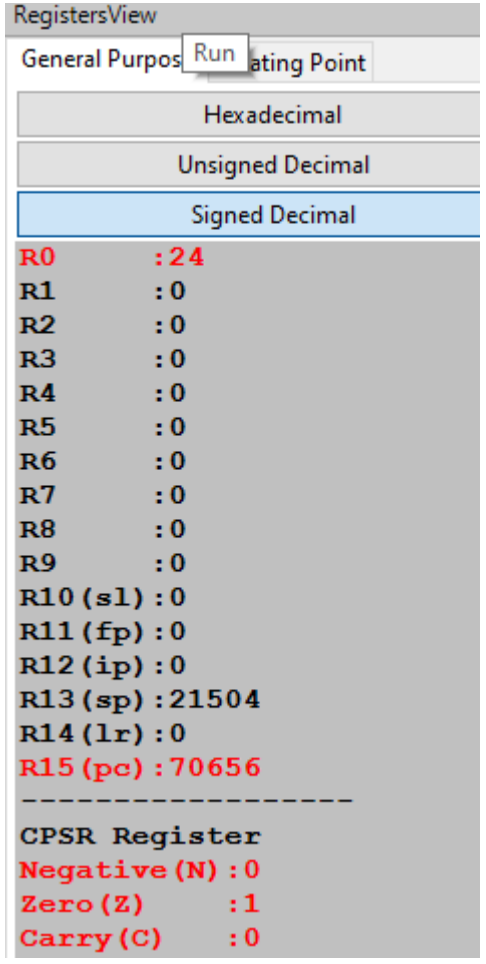


Microprocessor & Computer Architecture (µpCA)

Test (TST) & Test Equivalence (TEQ)

What do you infer from the following program?

```
.text
MOV R0,#24
TST R0,#1
.end
```



Logical Operations

- Operations are:
  - AND
  - EOR
  - ORR
  - BIC
- Syntax:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2
- Examples:
  - AND r0, r1, r2
  - BICEQ r2, r3, #7
  - EORS r1,r3,r0

Logical Operations

- `AND  R0, R1, R2    @ R0 = R1 and R2`
- `ORR  R0, R1, R2    @ R0 = R1 or  R2`
- `EOR  R0, R1, R2    @ R0 = R1 xor R2`
- `BIC  R0, R1, R2    @ R0 = R1 and (~R2)`

bit clear: `R2` is a mask identifying which bits of `R1` will be cleared to zero

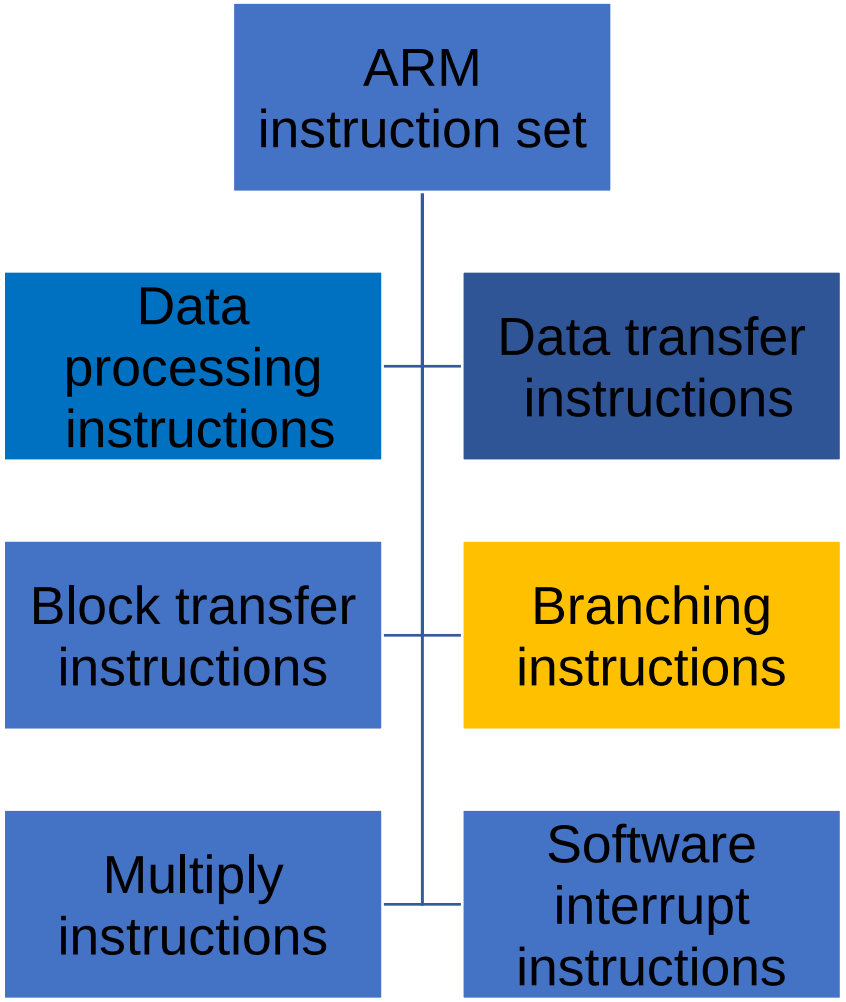
```
R1=0x11111111     R2=0x01100101

BIC R0, R1, R2

R0=0x10011010
```

Example: Logical Operation

```
MOV R0,#5
MOV R1,#6
AND R2,R0,R1 ; Logical AND
ORR R3,R0,R1 ; Logical OR
EOR R4,R0,R1 ; Logical XOR
MVN R5,R0    ; Complemented value is moved
to R5
```

```
R0      :5
R1      :6
R2      :4
R3      :7
R4      :3
R5      :-6
R6      :0
R7      :0
R8      :0
R9      :0
R10(sl):0
R11(fp):0
R12(ip):0
R13(sp):21504
R14(lr):0
R15(pc):70656
------------------
CPSR Register
Negative(N):0
Zero(Z)     :0
Carry(C)    :0
Overflow(V):0
IRQ Disable:1
FIQ Disable:1
Thumb(T)    :0
CPU Mode    :System
------------------
```

Next Class


ARM instruction set
Data processing instructions
Data transfer instructions
Block transfer instructions
Branching instructions
Multiply instructions
Software interrupt instructions

THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Barrel Shifter

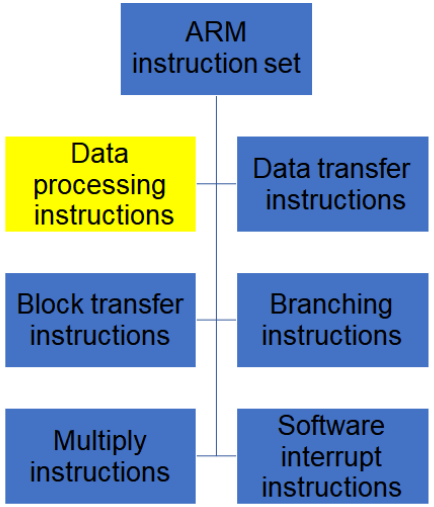
Dr. D. C. Kiran
Department of Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Syllabus

Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ~~ARM Program structure~~
- ~~ARM Instruction Format~~
- **ARM INSTRUCTION SET**
  - Data Processing Instructions
  - ~~**Data Movement Instruction**~~
  - ~~**Arithmetic Instruction**~~
  - ~~**Multiword Arithmetic**~~
  - **Barrel Shifter**

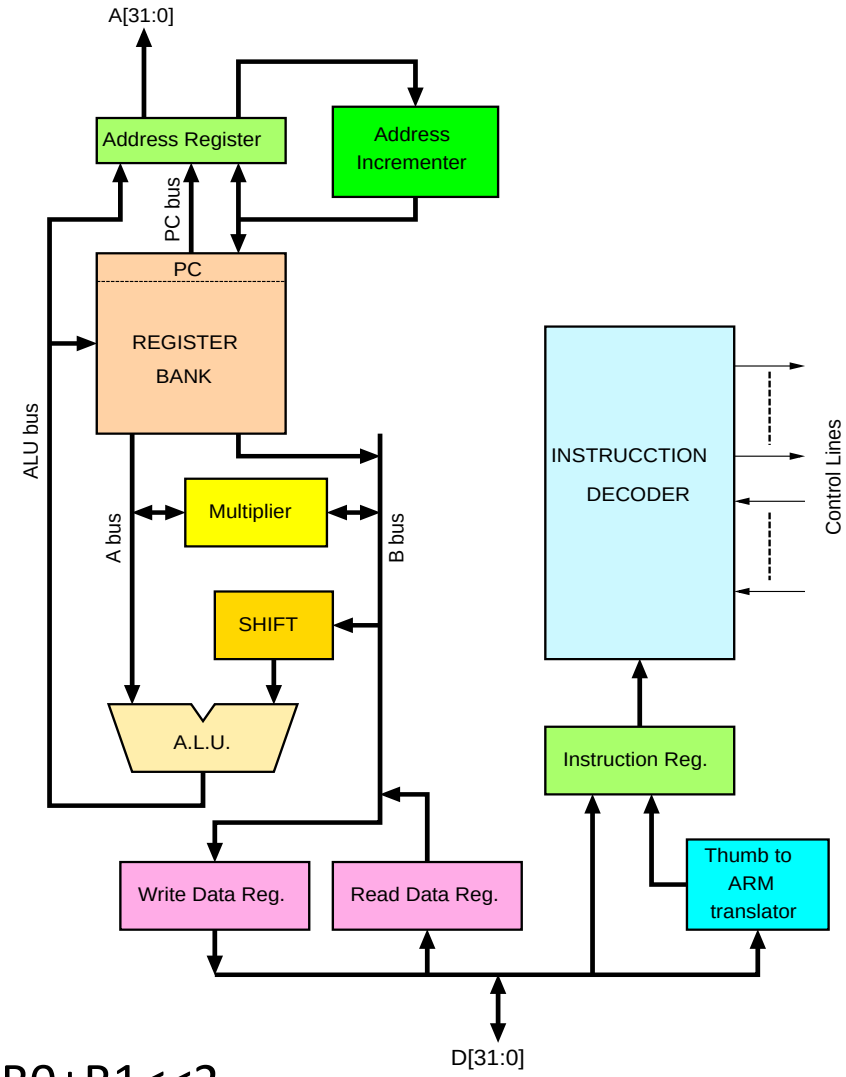


Microprocessor & Computer Architecture (µpCA)

Barrel Shifter

- The ARM doesn't have actual shift instructions.

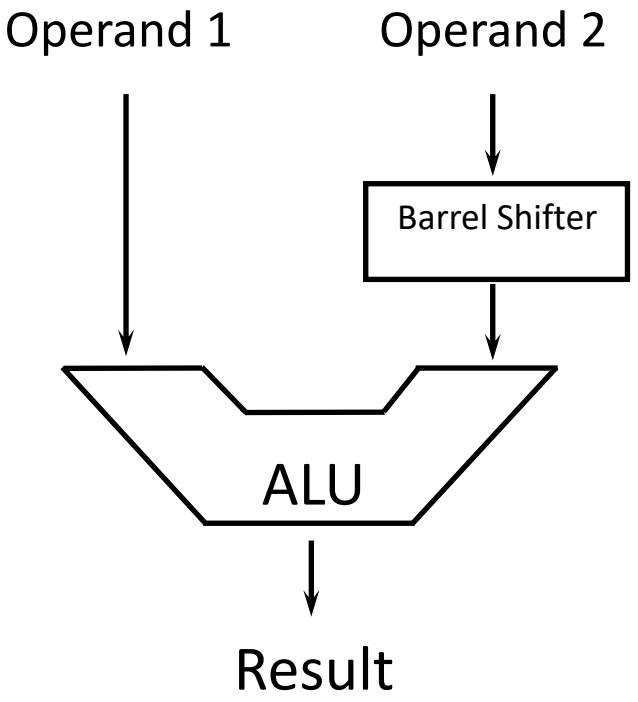
- Instead, it has a barrel shifter which provides a mechanism to carry out shifts as part of other instructions.



- Example:

  ADD R3,R0,R1, LSL#2 // R3=R0+R1<<2

Barrel Shifter

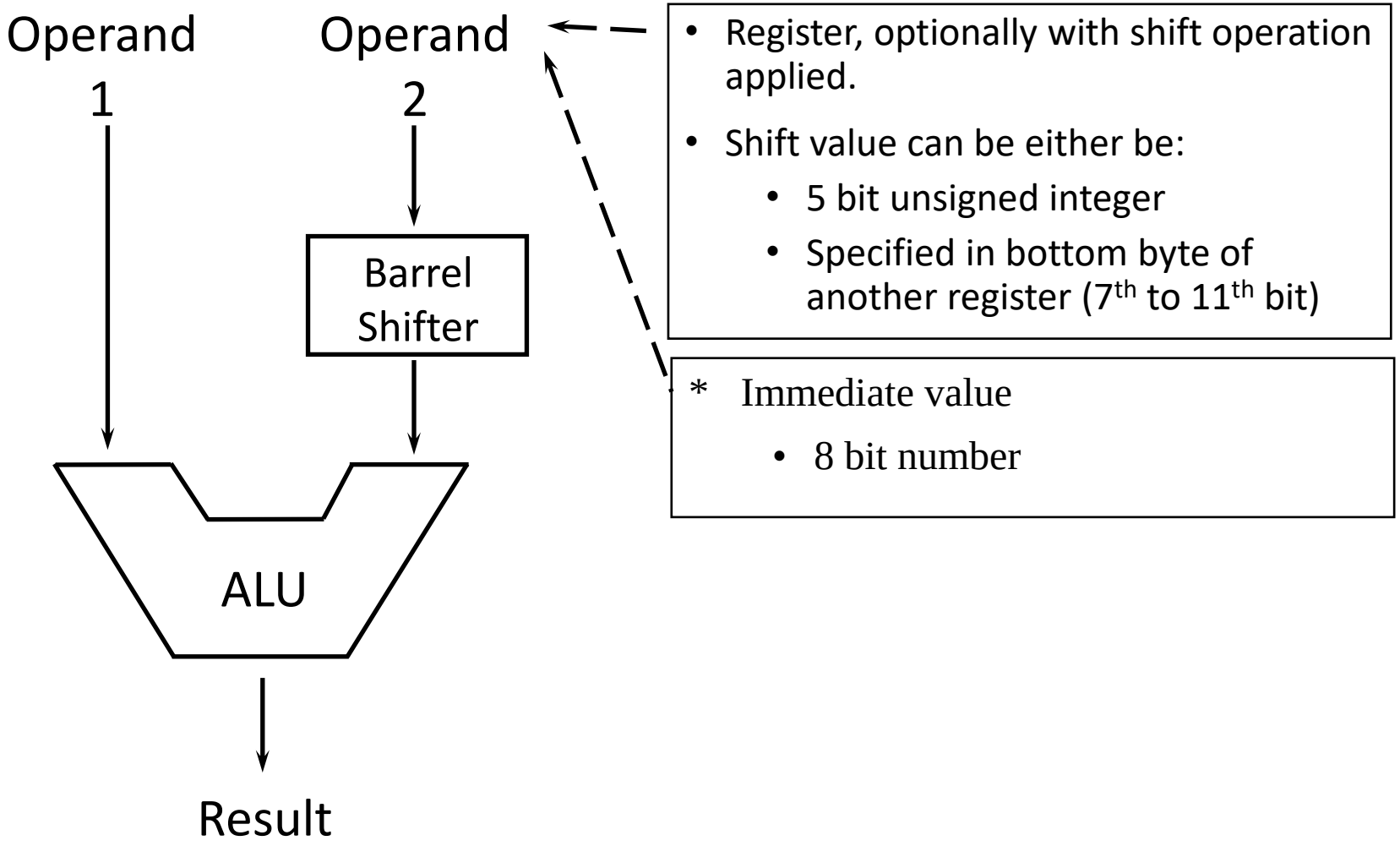


Possible Shift Operations:

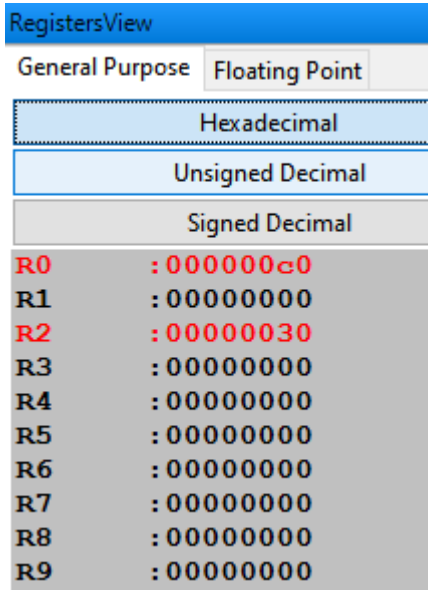
LSL: Left Shift
LSR: Right Shift
ASR: Arithmetic Right Shift
ROR: Rotate Right
RRX: Rotate Right Extended

- If no shift is specified then a default shift is applied: LSL #0
  - i.e. barrel shifter has no effect on value in register.

Barrel Shifter



Logical shift left




```
MOV   R0, R2, LSL #2 @ R0:=R2<<2
                     @ R2 unchanged
```

Example:

00000000 00000000 00000000 00110000

Before R2=0x00000030

After R2=0x00000030

R0=0x000000C0

00000000 00000000 00000000 11000000

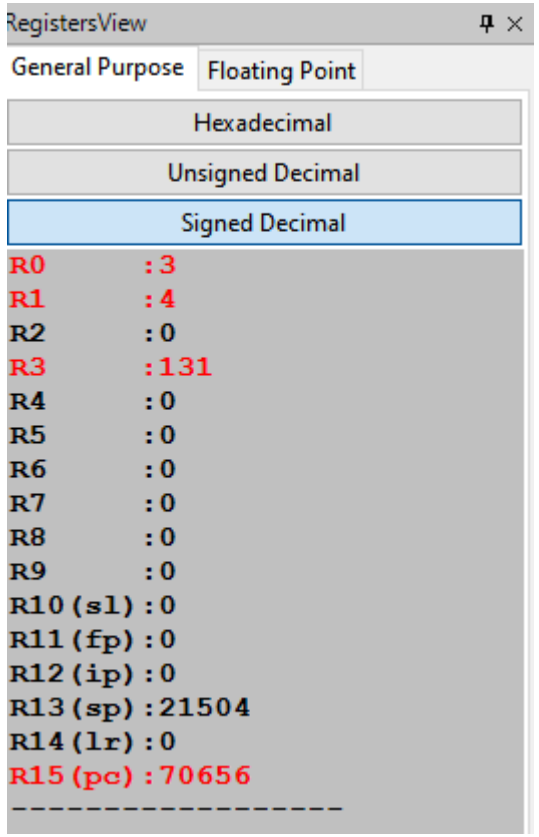
Barrel Shifter – Logical Left Shift (LSL)

- Shifts left by the specified amount (multiplies by powers of two) e.g.

LSL #5 = multiply by 32

```
.text
MOV R0, #3
MOV R1, #4
ADD R3,R0,R1,LSL#5
.end
```

R1 Before= 00000100
R1 After =10000000= 128



Output

R3 will have 131

Logical shift right (LSR)


```
MOV   R1, R2, LSR #2  @ R1:=R2>>2
                      @ R2 unchanged
```

Example:

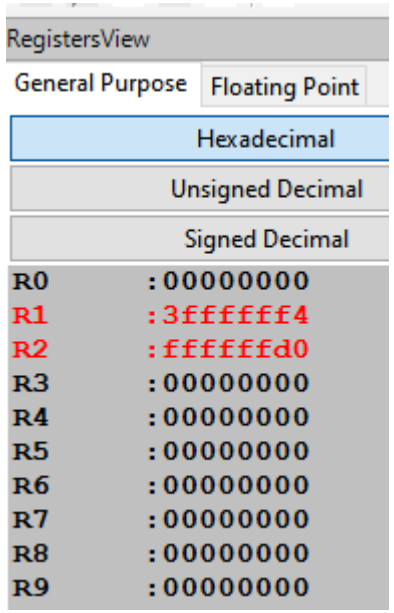
11111111 11111111 11111111 11010000

Before R2=0xffffffd0

After R1=0x3ffffff4

00111111 11111111 11111111 11110100

R2=0xffffffd0



Barrel Shifter - Right Shifts

```
.text
MOV R0, #3
MOV R1, #256
ADD R3,R0,R1,LSR #5
.end
```

Output

R3 will have 11

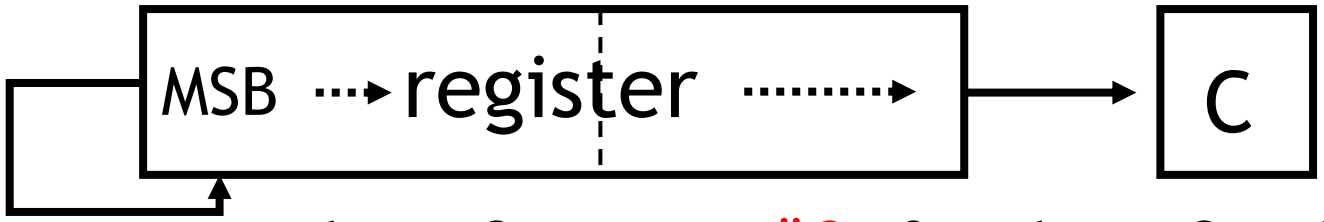
LSR is division 2^n
256/32

Before R1: 00000000 00000000 000001 0000 0000

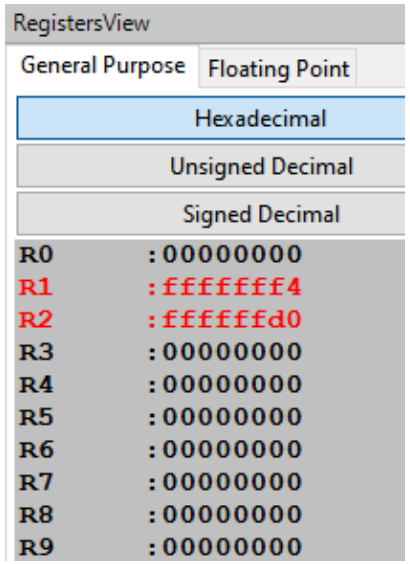
After Right Shift R1:00000000 00000000 000000 0000 1000 (8)
R0:00000000 00000000 000000 0000 0011 (3)

R3: 00000000 00000000 000000 0000 1011 (11)

Arithmetic shift right (ASR)




```
MOV  R1, R2, ASR #2  @ R1:=R2>>2
                     @ R2 unchanged
```



Example:

11111111 11111111 11111111 11010000

Before R2=0xffffffd0

After R1=0xfffffff4

11111111 11111111 11111111 11110100

R2=0xffffffd0

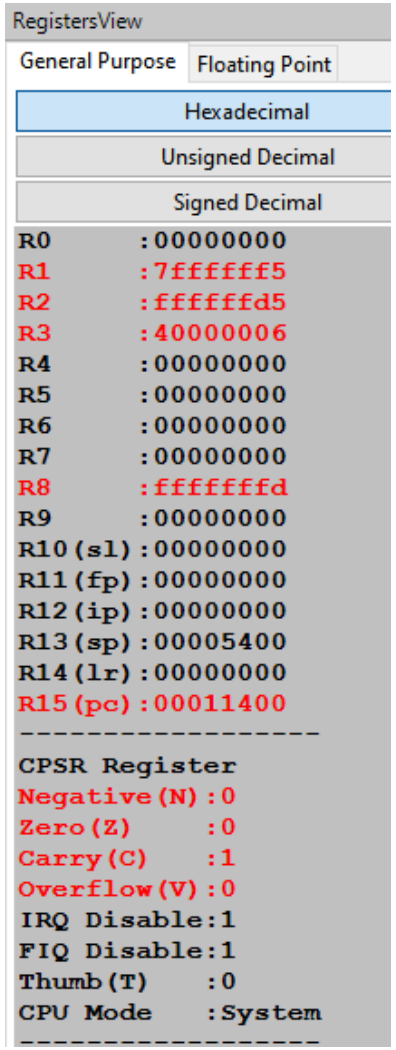
Rotate right (ROR)


```
MOV   R0, R2, ROR #2 @ R0:=R2 rotate
                     @ R2 unchanged
Example:
11111111 11111111 11111111 11010101
Before R2=0xffffffd5
After  R0=0x7fffffF5
(01111111 11111111 11111111 11110101)
       R2=0xffffffd4
```

Rotate Right Extended


```
MOV   R0, R2, RRX      @ R0:=R2 rotate
                       @ R2 unchanged
  Example: 0…0 0011 0001
Before R2=0x00000031, C=1
After  R0=0x80000018, C=1
          R2=0x00000031
```

```
.text
mov r2,#0xfffffffd5
adds r3,r2,#0x40000031
mov r1,r2,rrx, #2
.end
```



THINK ABOUT IT : MULTIPLICATION BY CONSTANT USING THE BARREL SHIFTER

- **Multiplication by 2^n (1,2,4,8,16,32..)**

  MOV Ra, Rb, LSL #n

- **Multiplication by 2^n+1 (3,5,9,17..)**

  ADD Ra,Ra,Ra,LSL #n

- **Multiplication by 2^n-1 (3,7,15..)**

  RSB Ra,Ra,Ra,LSL #n

- **Multiplication by 6**

  ADD Ra,Ra,Ra,LSL #1 ; multiply by 3

  MOV Ra,Ra,LSL#1 ; and then by 2

- **Multiply by 10 and add in extra number**

  ADD Ra,Ra,Ra,LSL#2 ; multiply by 5

  ADD Ra,Rc,Ra,LSL#1 ; multiply by 2 and add in next digit

Summary : Arithmetic Operations Addressing Mode

1. Register Direct Addressing: Operand values are in registers:

   ADD r3, r0, r1; r3=r0+r1

2. Immediate Addressing Mode: Operand value is within the instruction

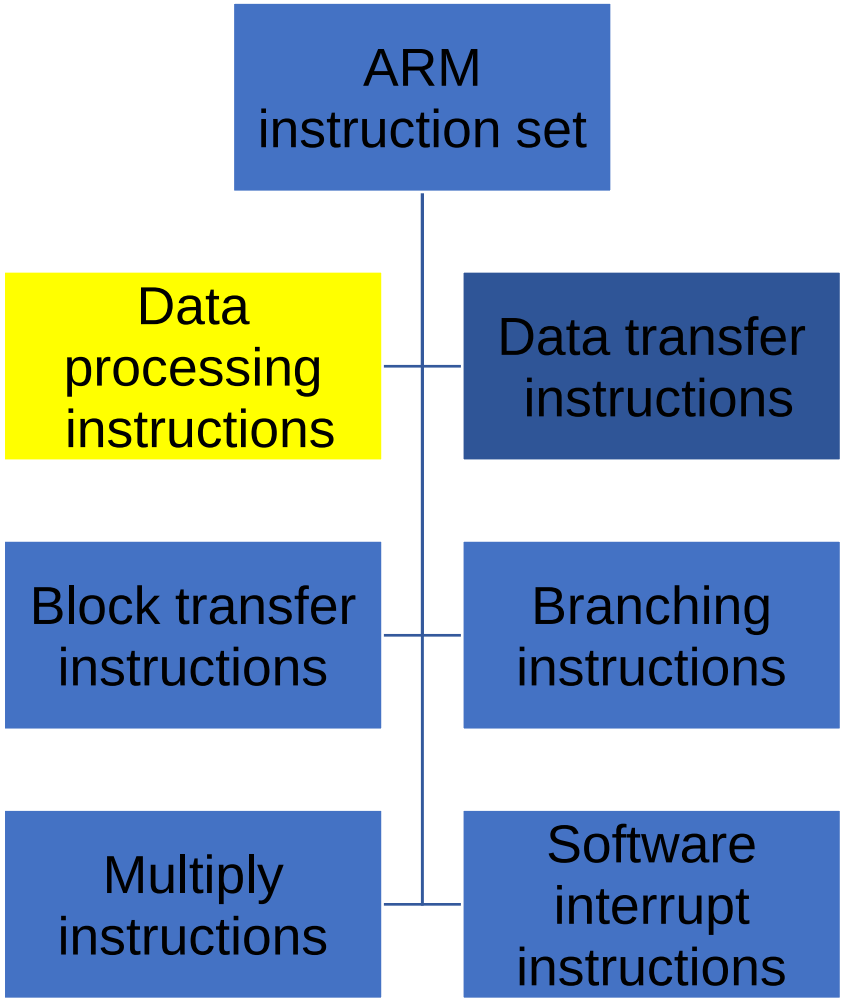
   ADD r3, r0, #7; r3=r0+7

   The number 7 is stored as part of the instruction

3. Register direct with shift or rotate (more next lecture)

   ADD r3, r0, r1, LSL#2; r3=r0+ r1<<2

Next Class: Logical and Comparison Instructions



THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

PES UNIVERSITY ONLINE

Microprocessor & Computer Architecture (µpCA)

Flow Control Instructions

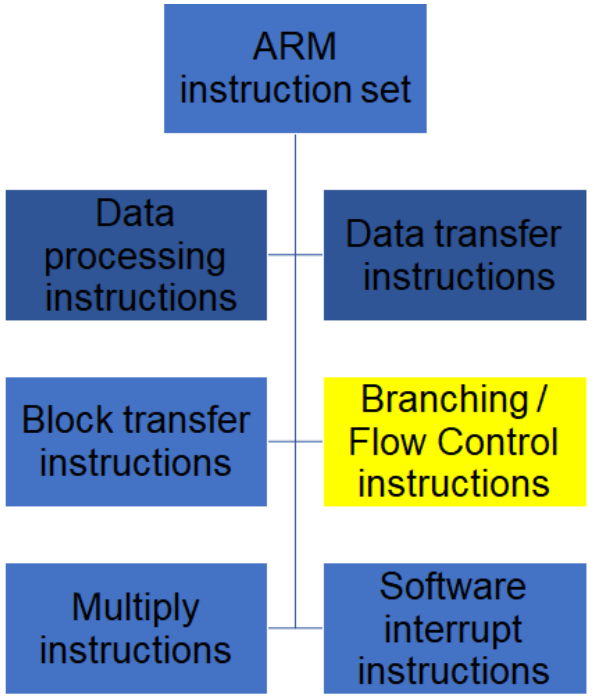
Dr. D. C. Kiran
Department of Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Syllabus

Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ~~ARM Program structure~~
- ~~ARM Instruction Format~~
- **ARM INSTRUCTION SET**
  - ~~Data Processing Instructions~~
  - Branch Instructions



Flow Control Instructions

Syntax: B{<cond>} Label
BL{<cond>} Label
BX{<cond>} Rm
BLX{<cond>} Rm

B	Branch	Program Counter = Label
BL	Branch & Link	**Step1:** PC will be copied to R14 the Link Register (LR) before branch is taken. **Step2:** Program Counter = Label
BX	Branch Exchange	Used for changing ARM to Thumb mode or from Thumb mode to ARM mode. Reference
BLX	Branch Exchange with link	

Flow Control Instructions

- Branch instruction

```
            B   label
            …
label:      …
```

- Conditional branches

```
            MOV   R0, #0
loop:         …
            ADD   R0, R0, #1
            CMP   R0, #10
            BNE   loop
```

Branch and Link

- **BL** instruction saves the return address to **R14** (lr)

```
        BL    sub       @ call sub
        CMP   R1, #5    @ return to here
        MOVEQ R1, #0
        …
sub:    …               @ sub entry point
        …
        MOV   PC, LR    @ return
```

Branch and Link (Nested Procedure Call)

```
        BL      sub1        @ call sub1
```

use stack to save/restore the return address and registers

```
sub1:   STMFD R13!, {R0-R2,R14}
        BL      sub2
        …
        LDMFD R13!, {R0-R2,PC}
```

```
sub2:   …
        MOV     PC, LR
```

Conditional Branch Options

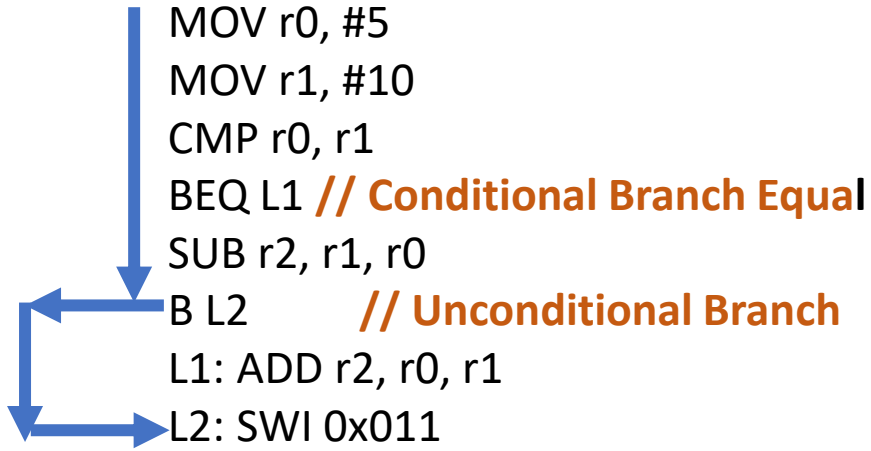
Branch	Interpretation	Normal uses
B BAL	Unconditional Always	Always take this branch Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC BLO	Carry clear Lower	Arithmetic operation did not give carry-out Unsigned comparison gave lower
BCS BHS	Carry set Higher or same	Arithmetic operation gave carry-out Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Microprocessor & Computer Architecture (µpCA)

Conditional Branch Example1

Compare the value of R0 and R1, add if R0 = R1, else subtract

```
.text
        MOV r0, #5
        MOV r1, #10
        CMP r0, r1
        BEQ L1 // Conditional Branch Equal
        SUB r2, r1, r0
        B L2        // Unconditional Branch
        L1: ADD r2, r0, r1
        L2: SWI 0x011
.end
```



```
R0      :5
R1      :10
R2      :5
R3      :0
R4      :0
R5      :0
R6      :0
R7      :0
R8      :0
R9      :0
R10(sl):0
R11(fp):0
R12(ip):0
R13(sp):21504
R14(lr):0
R15(pc):4124
------------------
CPSR Register
Negative(N):1
Zero(Z)     :0
Carry(C)    :0
Overflow(V):0
IRQ Disable:1
FIQ Disable:1
Thumb(T)    :0
CPU Mode    :System
------------------
```

Compare the value of R0 and R1, add if R0 = R1, else subtract

```
.text
        MOV r0, #10
        MOV r1, #10
        CMP r0, r1
        BEQ L1
        SUB r2, r1, r0
        B L2
L1:     ADD r2, r0, r1
L2:     SWI 0x011
.end
```

```
R0      :10
R1      :10
R2      :20
R3      :0
R4      :0
R5      :0
R6      :0
R7      :0
R8      :0
R9      :0
R10(sl):0
R11(fp):0
R12(ip):0
R13(sp):21504
R14(lr):0
R15(pc):4124
------------------
CPSR Register
Negative(N):0
Zero(Z)    :1
Carry(C)   :1
Overflow(V):0
IRQ Disable:1
FIQ Disable:1
Thumb(T)   :0
CPU Mode   :System
```

;Based on the value of the number in R0 :
;Store 1 in R1 if R0 is zero
;Store 2 in R1 if R0 is positive
;Store 3 in R1 if R0 is negative

```
Mov R0, #10
Cmp R0, #0
Moveq R1, #1
Beq L1
Movmi R1, #3
Bmi L1
Mov R1, #2
L1:
Swi 0x1011
```

R0 has the value 10

10!=0 . Update CPSR.z =0

Not Executed since CPSR.z=0

Branch not taken since CPSR.z=0

Not Executed, since CPSR.n=0

Branch Not taken, since CPSR.n=0

R1=2

```
;Based on the value of the number in R0 :
        ;Store 1 in R1 if R0 is zero
        ;Store 2 in R1 if R0 is positive
        ;Store 3 in R1 if R0 is negative

Mov R0, #-10        R0 has the value -10, CPSR.n=1
Cmp R0, #0          -10!=0 . Update CPSR.z =0
Moveq R1, #1        Not Executed since CPSR.z=0
Beq L1              Branch not taken since CPSR.z=0
Movmi R1, #3        Executed, since CPSR.n=1, R1=3
Bmi L1              Branch taken, since CPSR.n=1. Jump to L1
Mov R1, #2          Control will not reach this instruction
L1:
Swi 0x1011
```

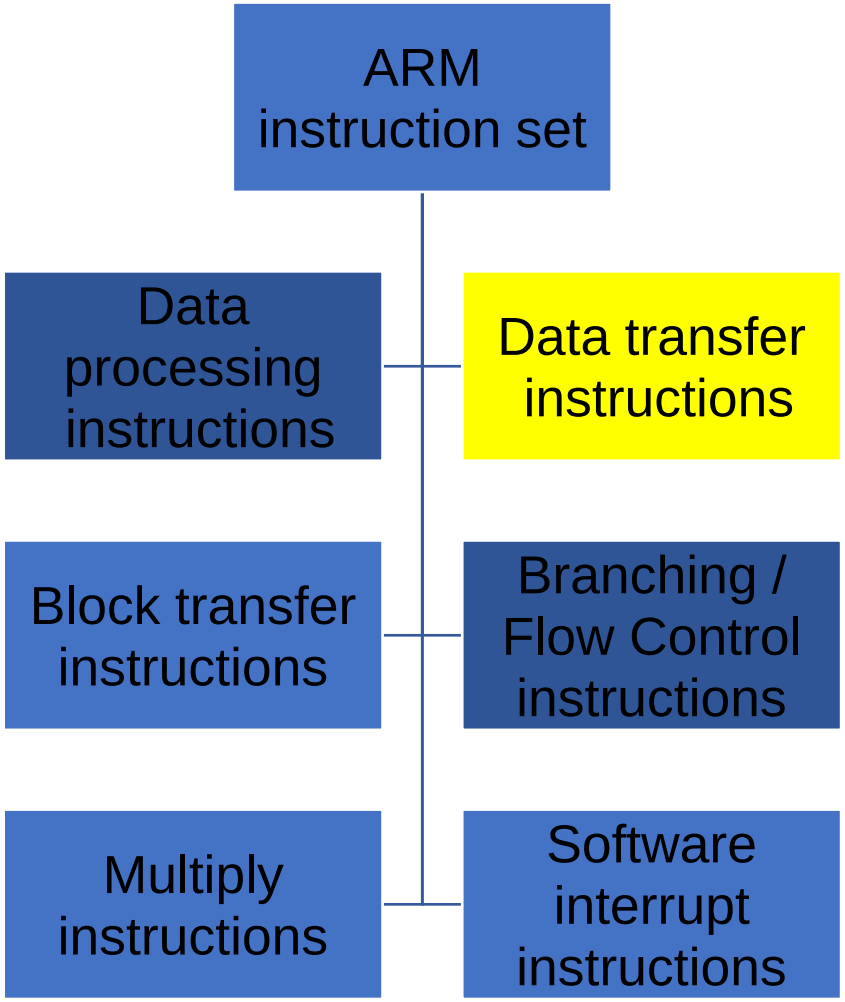
```
;Based on the value of the number in R0 :
        ;Store 1 in R1 if R0 is zero
        ;Store 2 in R1 if R0 is positive
        ;Store 3 in R1 if R0 is negative

Mov R0, #0      R0 has the value 0
Cmp R0, #0      0==0 . Update CPSR.z =1
Moveq R1, #1    Executed since CPSR.z=1, R1=1
Beq L1          Branch taken since CPSR.z=1
Movmi R1, #3    Control will not reach this instruction
Bmi L1          Control will not reach this instruction
Mov R1, #2      Control will not reach this instruction
L1:
Swi 0x1011
```

Microprocessor & Computer Architecture (µpCA)

Looping

;Factorial of a given number

```
mov r0, #5
mov r1, #1
l1:         mul r1, r0, r1
            subs r0, r0, #1
bne l1                          ;Comparison not equal or non-zero results
```


ARM instruction set
Data processing instructions
Data transfer instructions
Block transfer instructions
Branching / Flow Control instructions
Multiply instructions
Software interrupt instructions

THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

PES UNIVERSITY ONLINE

Microprocessor & Computer Architecture (µpCA)

Data Transfer Instructions

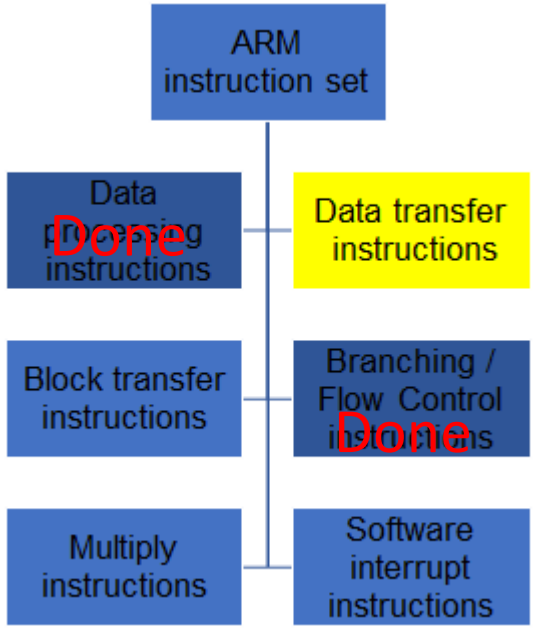
Dr. D. C. Kiran
Department of Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Syllabus

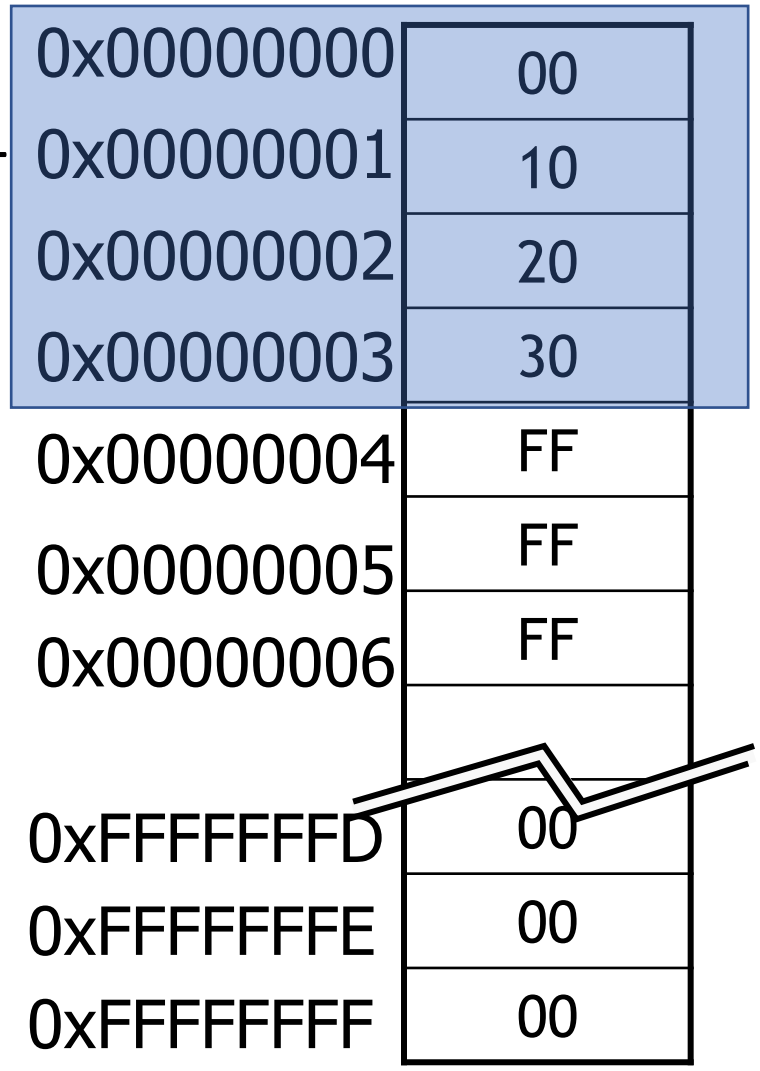
Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ~~ARM Program structure~~
- ~~ARM Instruction Format~~
- **ARM INSTRUCTION SET**
  - ~~Data Processing Instructions~~
  - ~~Flow Control Instructions~~
  - **Data Transfer Instructions**



Memory system

- Memory is a linear array of bytes addressed from 0 to 2^{32}-1
- Word, half-word, byte
- Little-endian

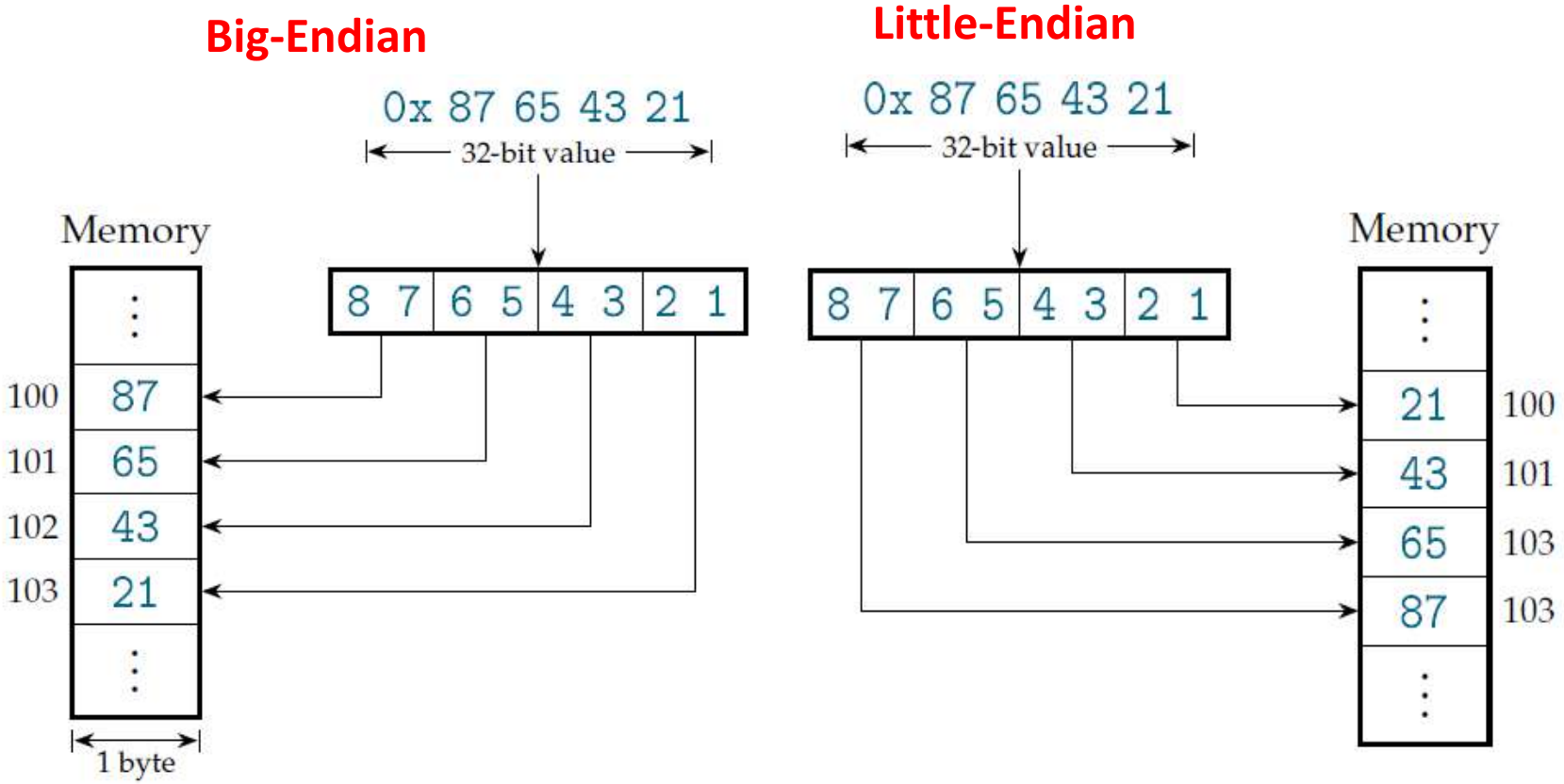


From ARMSIM

```
                        .text
00001000:E3A00013       mov r0,#19
00001004:E59F5018       ldr r5,=a
00001008:E5850000       str r0,[r5]
0000100C:E1A00000       mov r0,r0
00001010:E1A00000       mov r0,r0
00001014:E1A00000       mov r0,r0
00001018:E1A00000       mov r0,r0
0000101C:E1A00000       mov r0,r0
00001020:EF000011       swi 0x011
                        .data
00001028:               a: .word 0
```

Little-Endian vs. Big-Endian

- Little-endian: least-significant byte first
- Big-endian: most-significant byte first



- Move data between registers and memory
- Basic forms
  - Single register load/store
    - Ex: **LDR , STR**
  - Multiple register load/store or Block Transfer
    - Ex: **LDM, STM**

Single Register Load/Store

Syntax: <LDR/STR>{<cond>}{B} Rd, Addressing

LDR	Load word into register
STR	Save byte or word from register
LDRB	Load byte into register
STRB	Save byte from Register

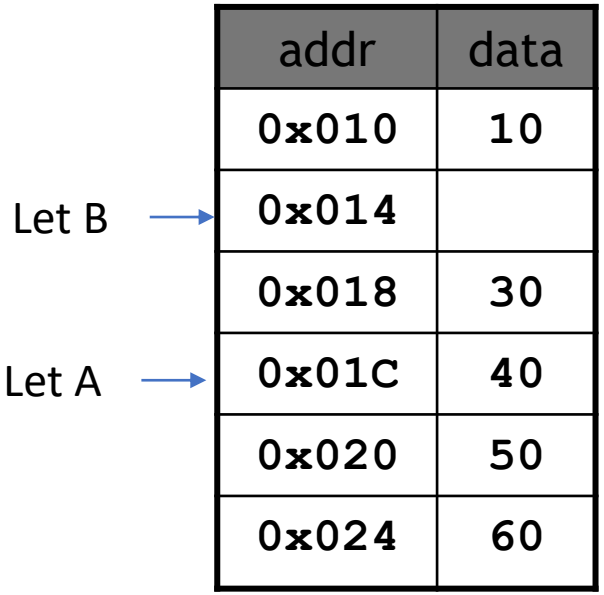
Single Register Load/Store

Syntax: LDR{<cond>}SB/H/SH Rd, Addressing
STR{<cond>}H Rd, Addressing

LDRH	Load half word into register
STRH	Save half word from a register
LDRSB	Load signed byte into register
LDRSH	Load signed halfword into a Register

No **`STRSB/STRSH`** since **`STRB/STRH`** stores both signed/unsigned ones

Copy A=B




```
LDR R0=A   ;Where a is a variable i.e address is copied to Register R0= 0x1C

LDR R5, [R0]    ; Copying the data in the address in R0 to R5=40

LDR R3=B   ;Where b is a variable i.e address is copied to Register R3 = 0x14
STR R5,[R3]  ; Store 40 in the address specified in R3=?
```

Copy A=B

	addr	data
	0x010	10
Let B →	0x014	
	0x018	30
Let A →	0x01C	40
	0x020	50
	0x024	60

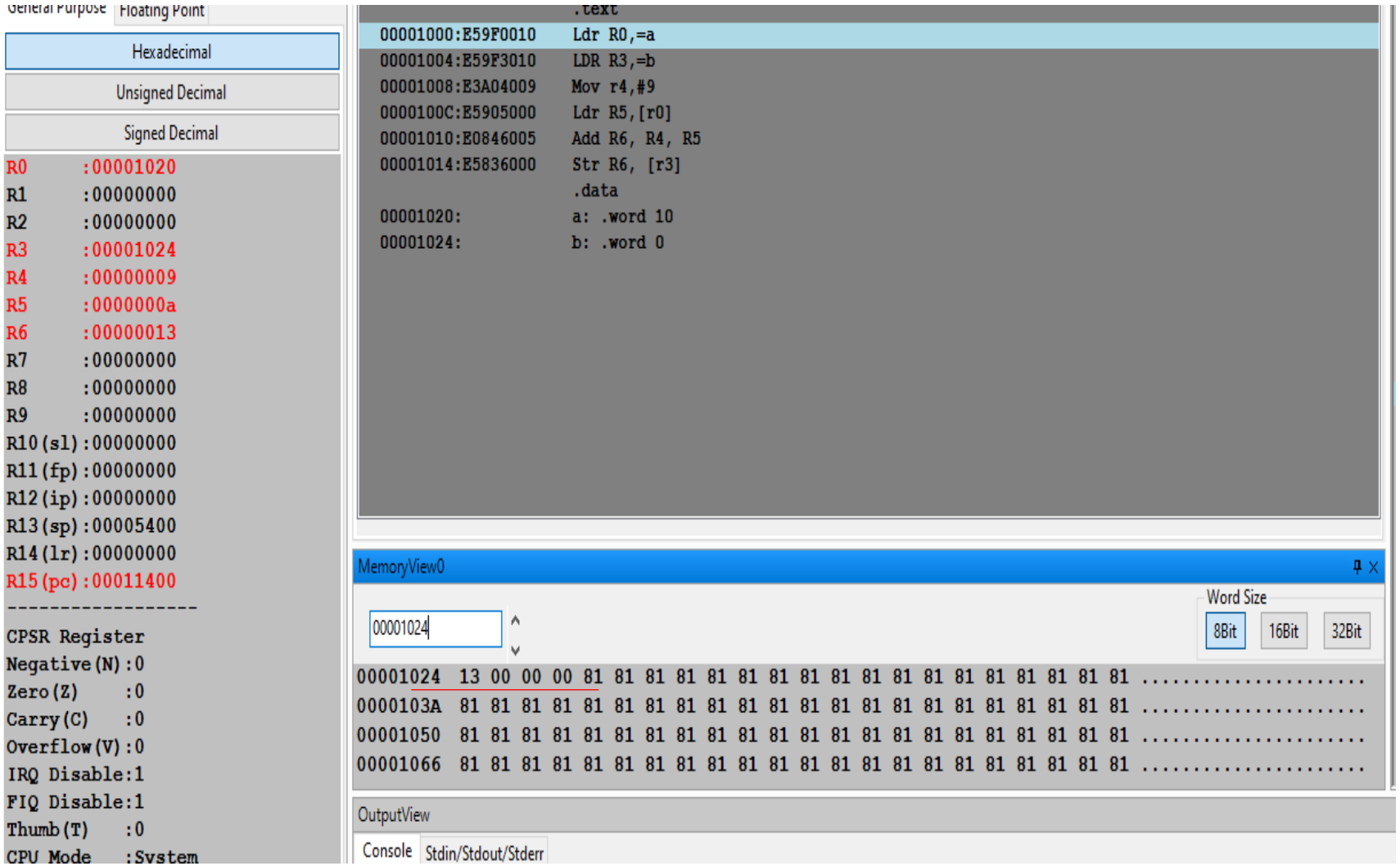
```
LDR R0=A   ;Where a is a variable i.e address is copied to Register R0= 0x1C

LDR R5, [R0]    ; Copying the data in the address in R0 to R5=40

LDR R3=B   ;Where b is a variable i.e address is copied to Register R3 = 0x14
STR R5,[R3]  ; Store 40 in the address specified in R3=40
```

LDR and STR Example B=A+9

```
.text
LDR R0, =A
LDR R3,=B
MOV R4,#9
LDR R5, [R0]
ADD R6, R4, R5
STR R6, [R3]
.data
A: .word 10
B: word 0
```

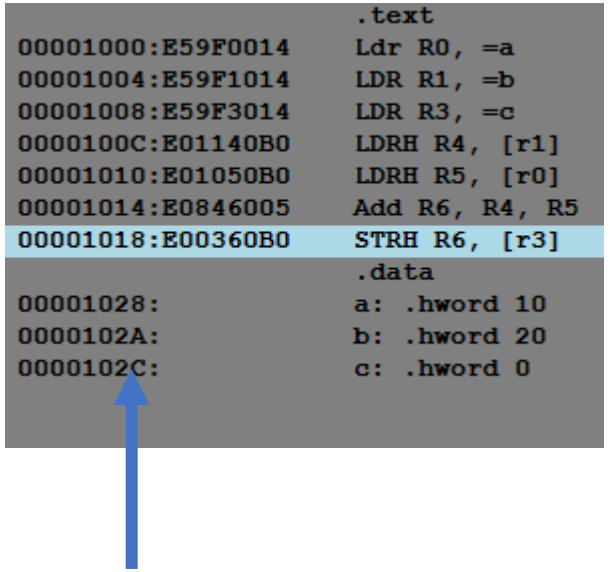
- LDR R0, =A → Address of A is copied to R0
- LDR R3,=B → Address of B is copied to R3
- MOV R4,#9 → Number 9 is copied to R4
- LDR R5, [R0] → Value in the A i.e 10 is copied to R5
- ADD R6, R4, R5 → 10+9=19 copy result to R6
- STR R6, [R3] → 19 is stored in the address in the register [R3], ie. B=19

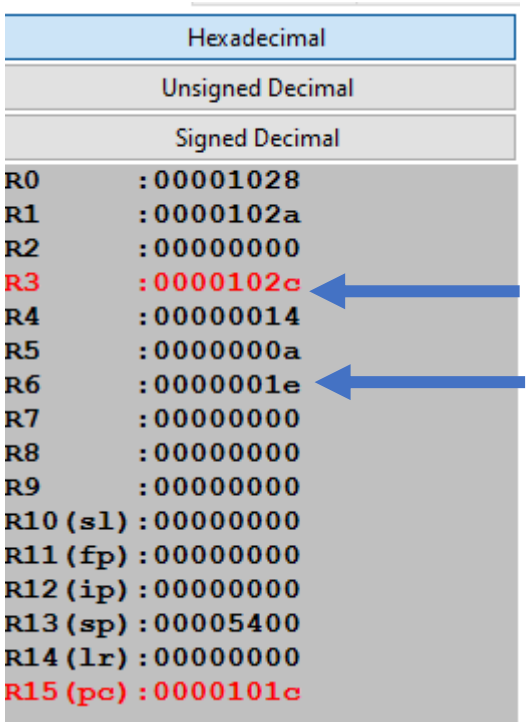
Microprocessor & Computer Architecture (µpCA)

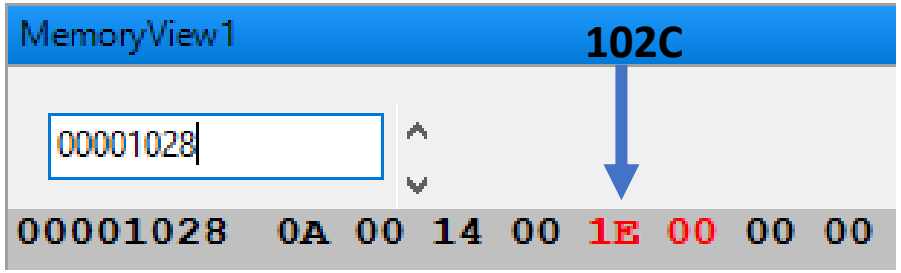


Example 2: C=A+B (Half Word)

```
.text
LDR R0, =a
LDR R1, =b
LDR R3, =c
LDRH R4, [R1]
LDRH R5, [R0]
ADD R6, R4, R5
STRH R6, [R3]
.data
a: .hword 10
b: .hword 20
c: .hword 0
```







Microprocessor & Computer Architecture (µpCA)

Example 3: String (BYTE)

```
.text
Ldr         R0, =mystring
Swi 0x02
Swi 0x11
.data
mystring:.asciz "HELLO PESU"
```

jstring.s

```
                        .text
00001000:E59F0004       Ldr     R0, =mystring
00001004:EF000002       Swi 0x02
00001008:EF000011       Swi 0x11
                        .data
00001010:               mystring:.asciz "HELLO PESU"
```

Hexadecimal
Unsigned Decimal
Signed Decimal

```
R0      :00001010
R1      :00000000
R2      :00000000
R3      :00000000
R4      :00000000
R5      :00000000
R6      :00000000
R7      :00000000
R8      :00000000
R9      :00000000
R10(sl) :00000000
R11(fp) :00000000
R12(ip) :00000000
R13(sp) :00005400
R14(lr) :00000000
R15(pc) :00001008
----------------
```

MemoryView1

00001010

```
00001010  48 45 4C 4C 4F 20 50 45 53 55 00 00 81
00001027  81 81 81 81 81 81 81 81 81 81 81 81 81
```

OutputView

Console | Stdin/Stdout/Stderr

```
HELLO PESU
```

Addressing or Indexing

- Pre Indexing Without Write Back
- **Syntax:** LDR Rd, [Rn,OFFSET]
- Pre Indexing With Write Back

  Syntax: LDR Rd, [Rn,OFFSET]!
- Poste Indexing

  Syntax: LDR Rd, [Rn] ,OFFSET

THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

PES UNIVERSITY ONLINE

Microprocessor & Computer Architecture (µpCA)

Data Transfer Instructions

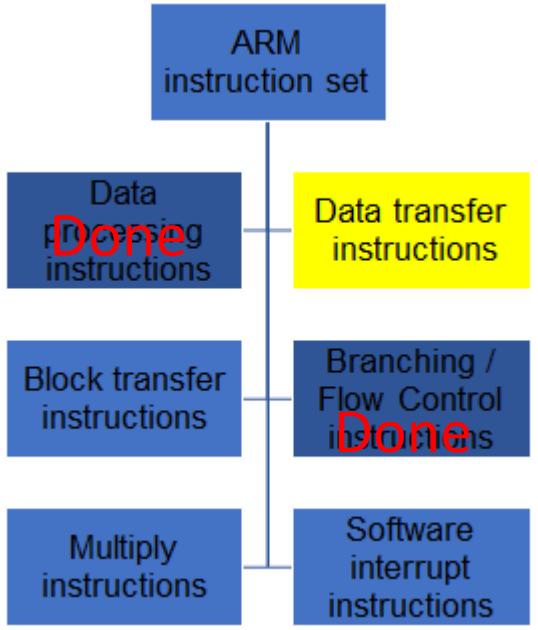
Dr. D. C. Kiran
Department of Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Syllabus

Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ~~ARM Program structure~~
- ~~ARM Instruction Format~~
- **ARM INSTRUCTION SET**
  - ~~Data Processing Instructions~~
  - ~~Flow Control Instructions~~
  - **Data Transfer Instructions**
    - ~~LDR & STR~~
    - **Indexing**



- Pre Indexing Without Write Back
- **Syntax:** LDR Rd, [Rn,OFFSET]
- Pre Indexing With Write Back

  Syntax: LDR Rd, [Rn,OFFSET]!
- Poste Indexing

  Syntax: LDR Rd, [Rn] ,OFFSET

- Memory is addressed by a register and an offset.

```
LDR   R0, [R1]  @ mem[R1]
```

- Three ways to specify offsets:
  - **Immediate**

```
LDR   R0,[R1,#4]      @ mem[R1+4]
```

  - **Register**

```
LDR   R0,[R1,R2]      @ mem[R1+R2]
```

  - **Scaled Register**

```
LDR   R0,[R1,R2,LSL #2] @ mem[R1+4*R2]
```

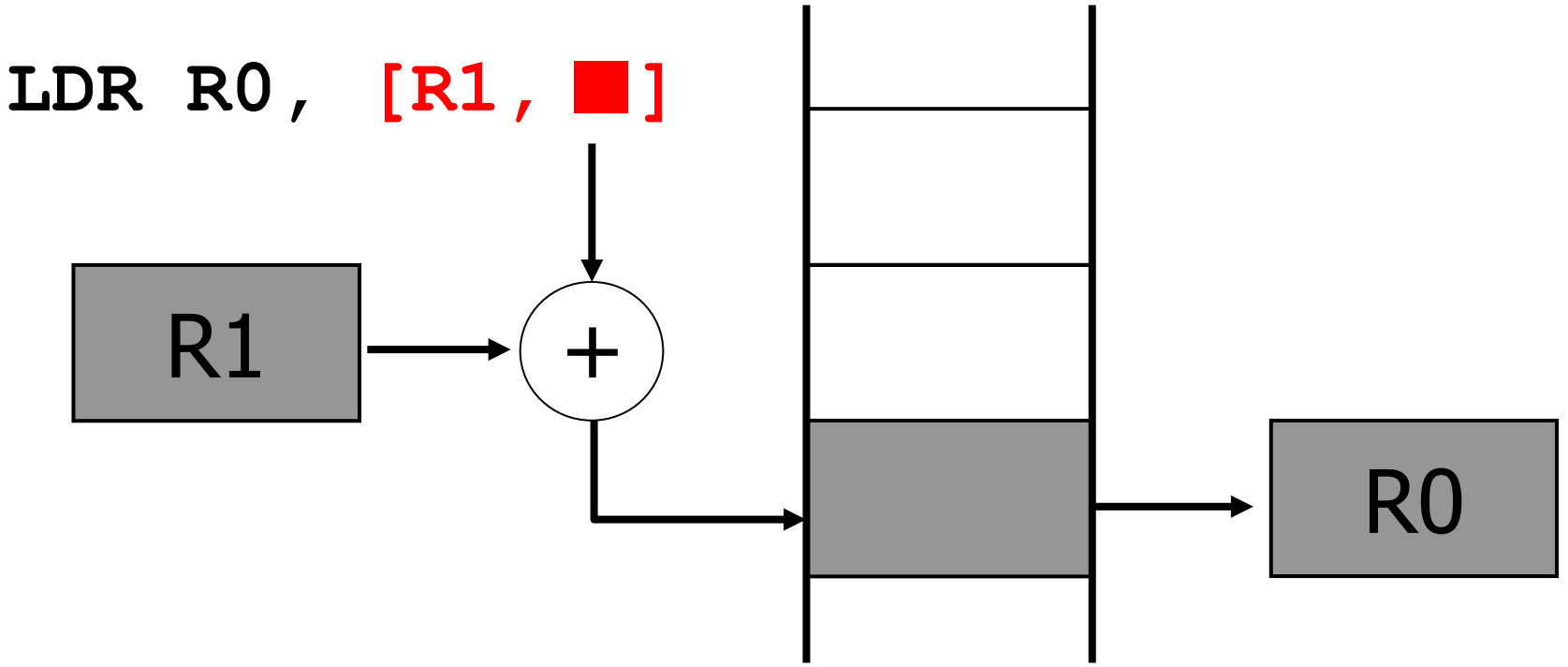
- Pre-index addressing (`LDR R0, [R1, #4]`)
  without a writeback.

- Auto-indexing addressing (`LDR R0, [R1, #4]!`)
  Pre-index with writeback
  calculation before accessing with a writeback

- Post-index addressing (`LDR R0, [R1], #4`)
  calculation after accessing with a writeback

Pre-index addressing

```
LDR   R0, [R1, #4]     @ R0=mem[R1+4]
                       @ R1 unchanged
```

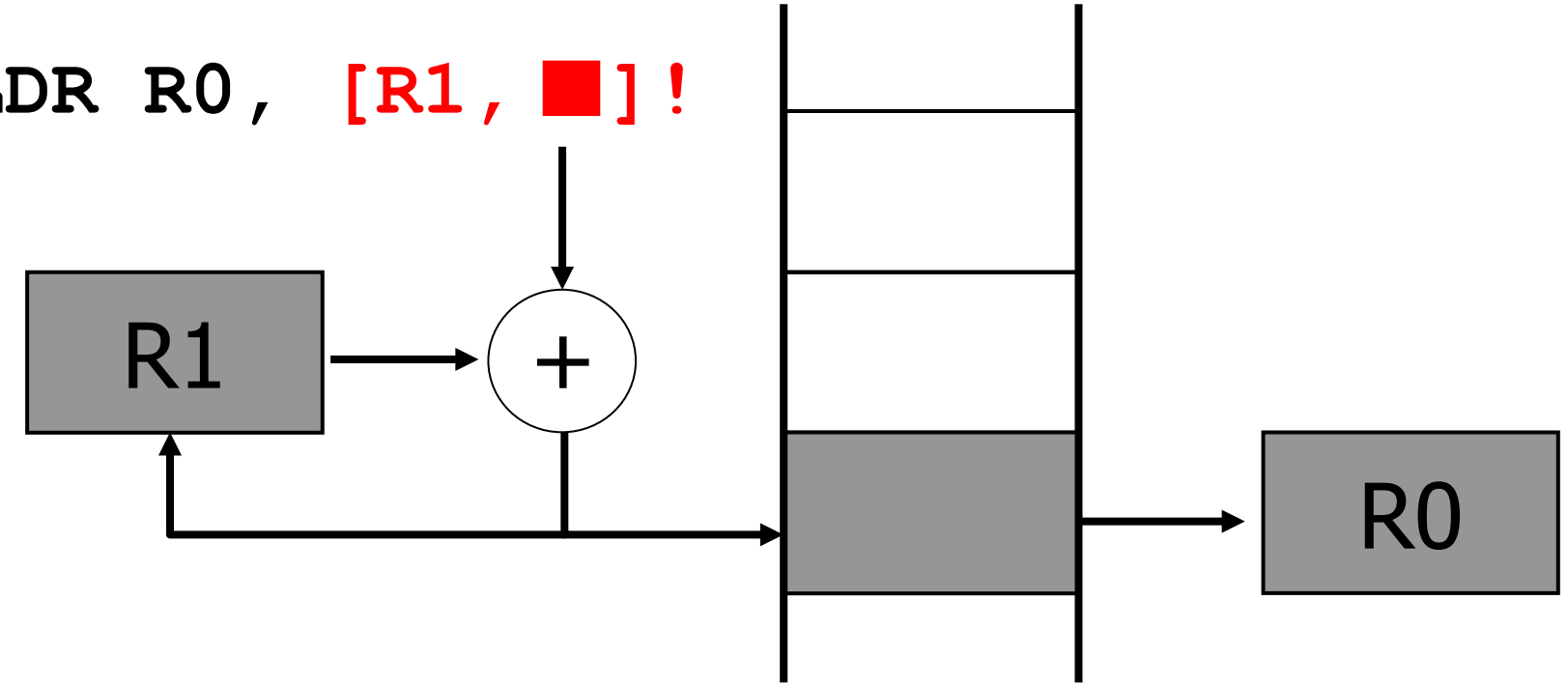


Auto-indexing addressing

```
LDR   R0, [R1, #4]!   @ R0=mem[R1+4]
                      @ R1=R1+4
```

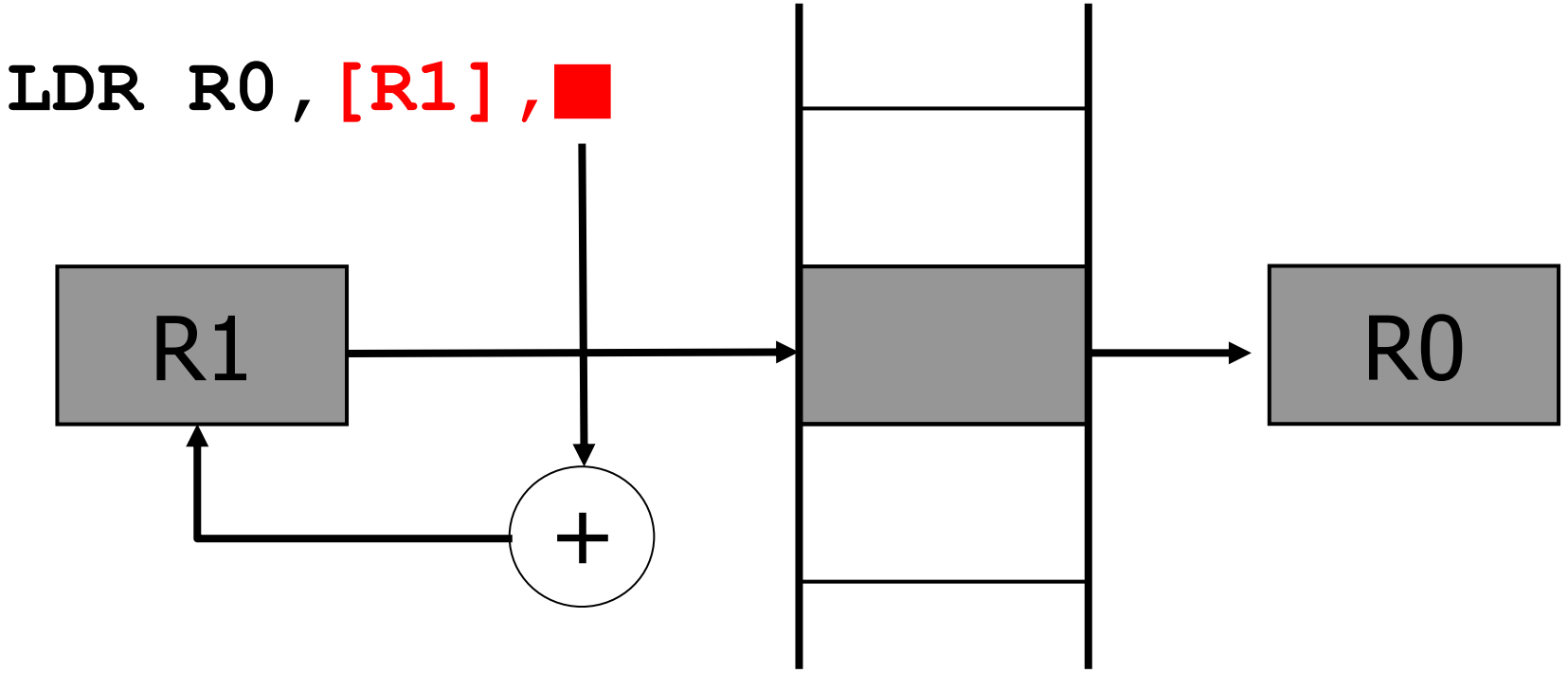
No extra time; Fast;

LDR R0, [R1, ■]!



Post-index addressing

```
LDR   R0, [R1], #4        @ R0=mem[R1]
                          @ R1=R1+4
```



Microprocessor & Computer Architecture (µpCA)

Pre-indexed addressing without Write Back or Auto Indexing

```
                        .text
00001000:E59F1000       LDR R1,=A
00001004:E5910004       LDR R0,[R1,#4]
                        .data
0000100C:               A: .word 10,20,30,40
```

RegistersView

General Purpose | Floating Point

Hexadecimal | Unsigned Decimal | Signed Decimal

```
R0        :00000014
R1        :0000100c
R2        :00000000
R3        :00000000
R4        :00000000
R5        :00000000
R6        :00000000
R7        :00000000
R8        :00000000
R9        :00000000
R10 (sl)  :00000000
R11 (fp)  :00000000
R12 (ip)  :00000000
R13 (sp)  :00005400
R14 (lr)  :00000000
R15 (pc)  :00001008
```

R0 get the 0x100C+4th Value

R1 is not Updated

MemoryView1

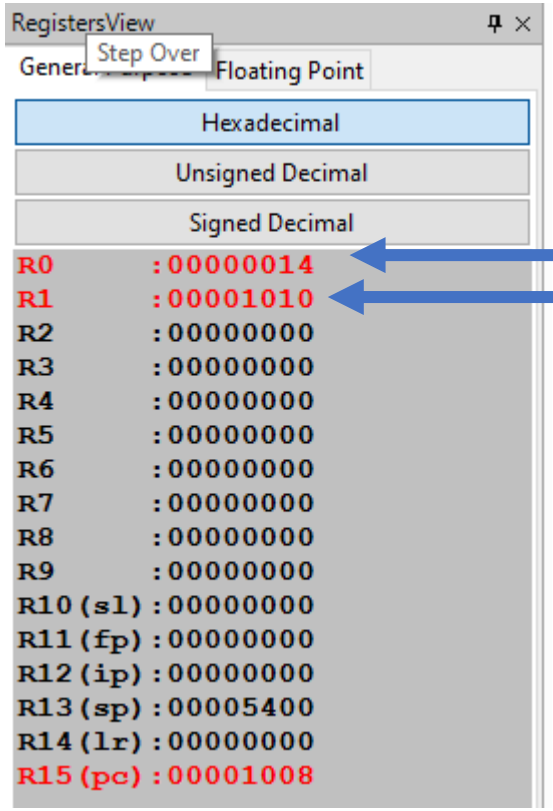
0000100C

```
0000100C  0000000A  00000014  0000001E  00000028
00001030  81818181  81818181  81818181  81818181
00001054  81818181  81818181  81818181  81818181
```

Microprocessor & Computer Architecture (µpCA)

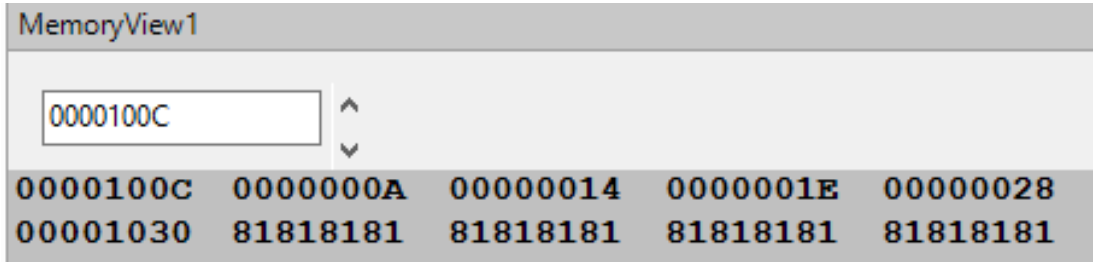
Pre-indexed addressing with Write Back or Auto Indexing-2

```
                        .text
00001000:E59F1000       LDR R1,=A
00001004:E5B10004       LDR R0,[R1,#4]!
                        .data
0000100C:               A: .word 10,20,30,40
```



R0 get the 0x100C+4th i.e 0x1010th Value

R1 is Updated to 0x1010



Microprocessor & Computer Architecture (µpCA)

Post-index addressing:2

```
                        .text
00001000:E59F1000       LDR R1,=A
00001004:E4910004       LDR R0,[R1],#4
                        .data
0000100C:               A: .word 10,20,30,40
```

RegistersView

General Purpose | Floating Point

Hexadecimal

Unsigned Decimal

Signed Decimal

```
R0      :0000000a
R1      :00001010
R2      :00000000
R3      :00000000
R4      :00000000
R5      :00000000
R6      :00000000
R7      :00000000
R8      :00000000
R9      :00000000
R10(sl) :00000000
R11(fp) :00000000
R12(ip) :00000000
R13(sp) :00005400
R14(lr) :00000000
R15(pc) :00001008
```

R0 get the 0x100Cth Value

R1 is Updated to 0x1010

MemoryView1

0000100C

```
0000100C  0000000A  00000014  0000001E  00000028
00001030  81818181  81818181  81818181  81818181
00001054  81818181  81818181  81818181  81818181
```

Next Session:

Block Transfer

- **LDRM**
- **STRM**

THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

PES UNIVERSITY ONLINE

Microprocessor & Computer Architecture (µpCA)

Block Transfer Instructions: Stack

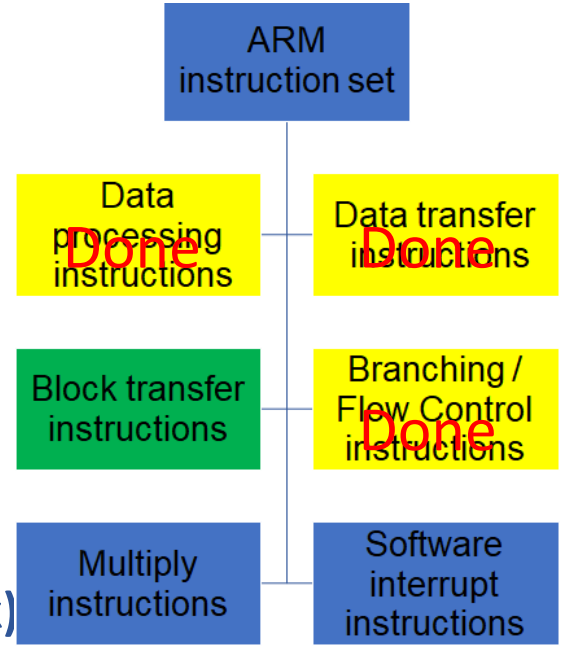
Dr. D. C. Kiran
Department of Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Syllabus

Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ~~ARM Program structure~~
- ~~ARM Instruction Format~~
- **ARM INSTRUCTION SET**
  - ~~Data Processing Instructions~~
  - ~~Flow Control Instructions~~
  - ~~Data Transfer Instructions~~
  - **Block Transfer Instructions (Stack)**



Block Transfer Instructions (Stack)

The Memory access can be in FILO fashion.

i.e Can be treated like STACK.

R13 is a stack pointer which will keep the address of TOP of the STACK.

Mainly used in Procedural Call.

Stack can grow upward or downward direction based on the MODE used by the user in the program.

Addr	data
0x1010	
0x1014	
0x1018	10
0x101C	20
0x1010	30
0x1020	40
0x1024	50
0x1028	60
0x102C	
0x1030	
0x1034	

Microprocessor & Computer Architecture (µpCA)

Block Transfer Instructions (Stack)

10 20 10 140 15 60 27

R13=0x1018

SP

Addr	data
0x1010	
0x1014	
0x1018	
0x101C	
0x1010	
0x1020	
0x1024	
0x1028	
0x102C	
0x1030	
0x1034	

Microprocessor & Computer Architecture (µpCA)

Block Transfer Instructions (Stack)

10 20 75 140 15 60 27

R13=0x101C

SP

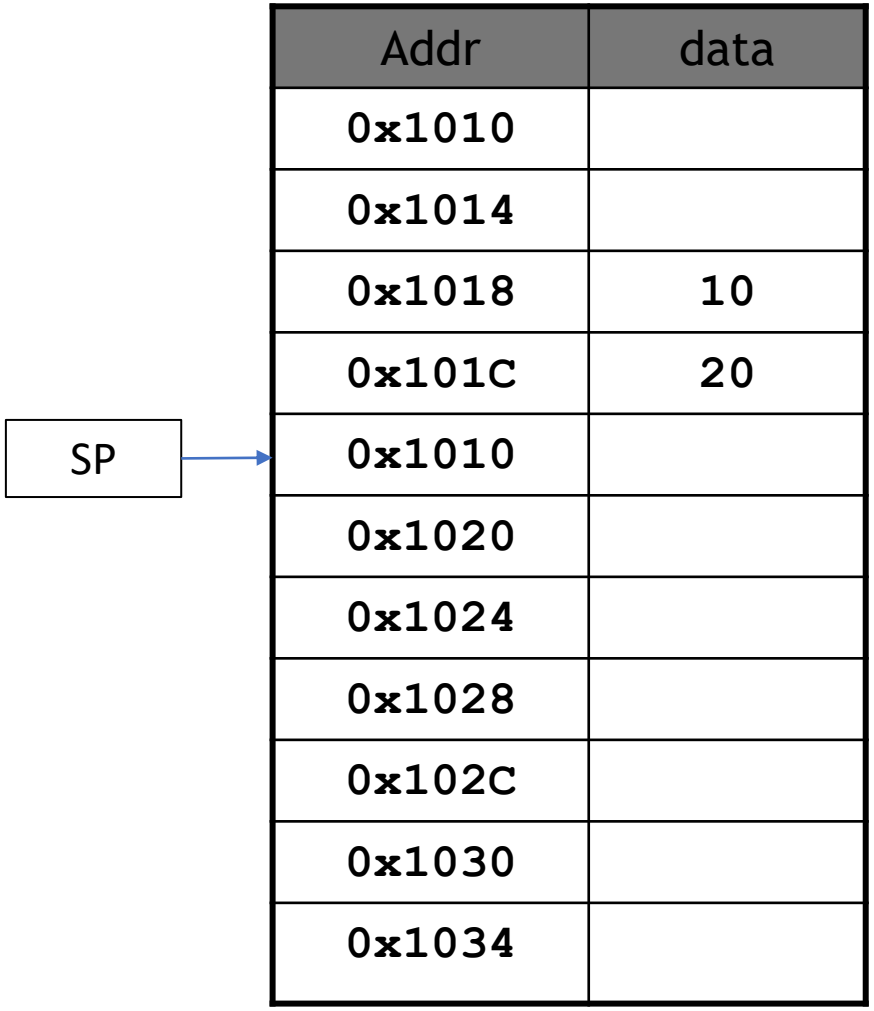
Addr	data
0x1010	
0x1014	
0x1018	10
0x101C	
0x1010	
0x1020	
0x1024	
0x1028	
0x102C	
0x1030	
0x1034	

Microprocessor & Computer Architecture (µpCA)

Block Transfer Instructions (Stack)

10 20 75 140 15 60 27

R13=0x1010



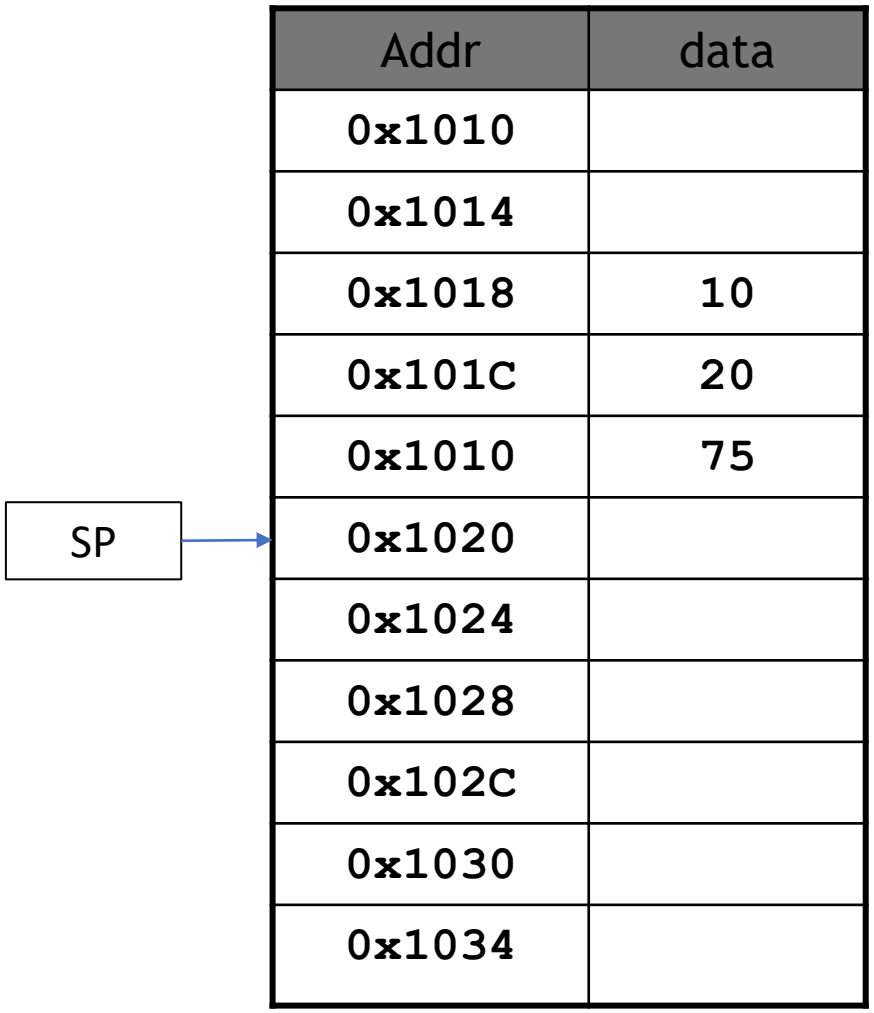
Addr	data
0x1010	
0x1014	
0x1018	10
0x101C	20
0x1010	
0x1020	
0x1024	
0x1028	
0x102C	
0x1030	
0x1034	

Microprocessor & Computer Architecture (µpCA)

Block Transfer Instructions (Stack)

10 20 75 140 15 60 27

R13=0x1020



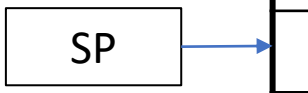
Addr	data
0x1010	
0x1014	
0x1018	10
0x101C	20
0x1010	75
0x1020	
0x1024	
0x1028	
0x102C	
0x1030	
0x1034	

Microprocessor & Computer Architecture (µpCA)

Block Transfer Instructions (Stack)

10 20 75 140 15 60 27

R13=0x1024



Addr	data
0x1010	
0x1014	
0x1018	10
0x101C	20
0x1010	75
0x1020	140
0x1024	
0x1028	
0x102C	
0x1030	
0x1034	

Block Transfer Instructions (Stack)

Syntax:

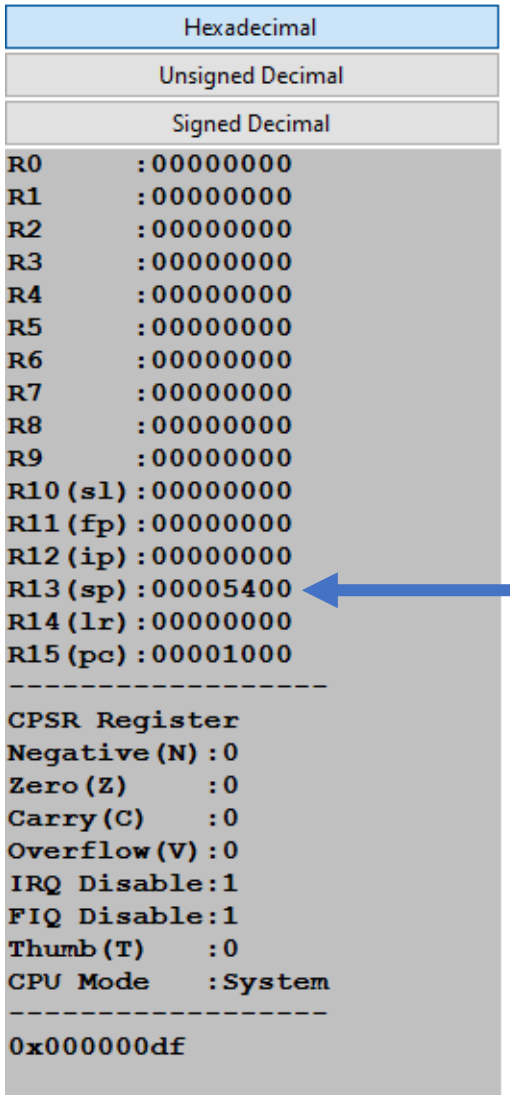
<LDM/STM> <Addressing Mode>R13 {!},Registers

Addressing Mode	=LDM	=STM
Full ascending (FA)	LDMDA	STMIB
Full descending (FD)	LDMIA	STMDB
Empty ascending (EA)	LDMDB	STMIA
Empty descending (ED)	LDMIB	STMDA

STM is used to PUSH on to the STACK
LDM is used to POP from the STACK

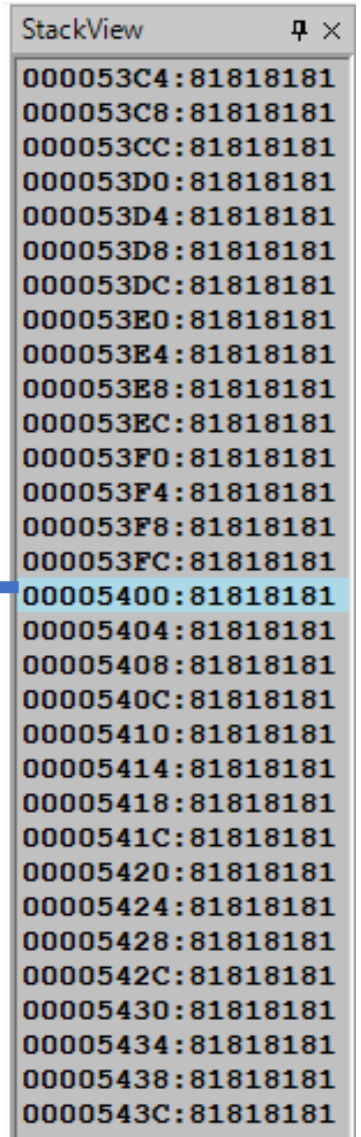
Microprocessor & Computer Architecture (µpCA)

Block Transfer Instructions (Stack)



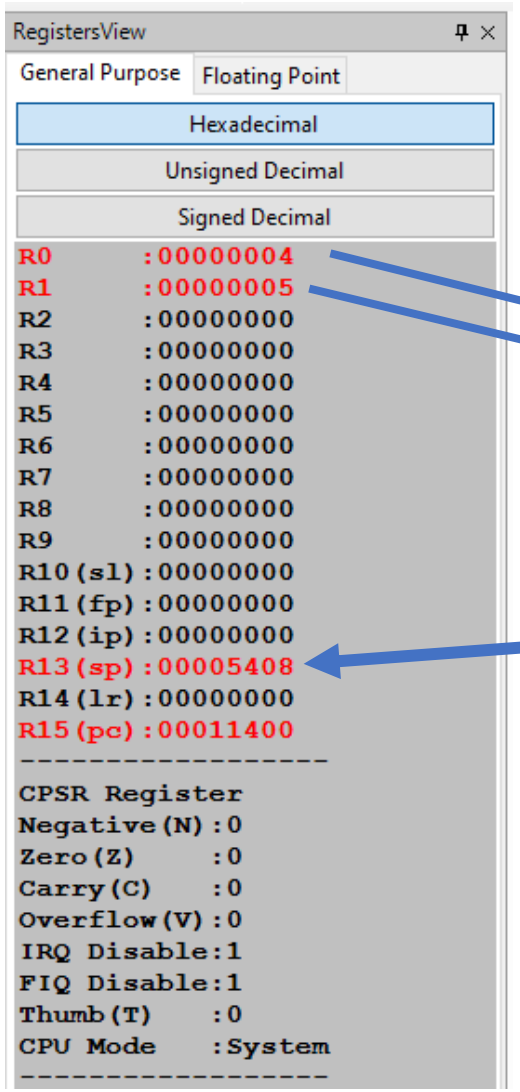

```
.text
MOV R0,#4
MOV R1,#5
STMEA R13!, {R0,R1}
.end
```

Initially Stack is Empty



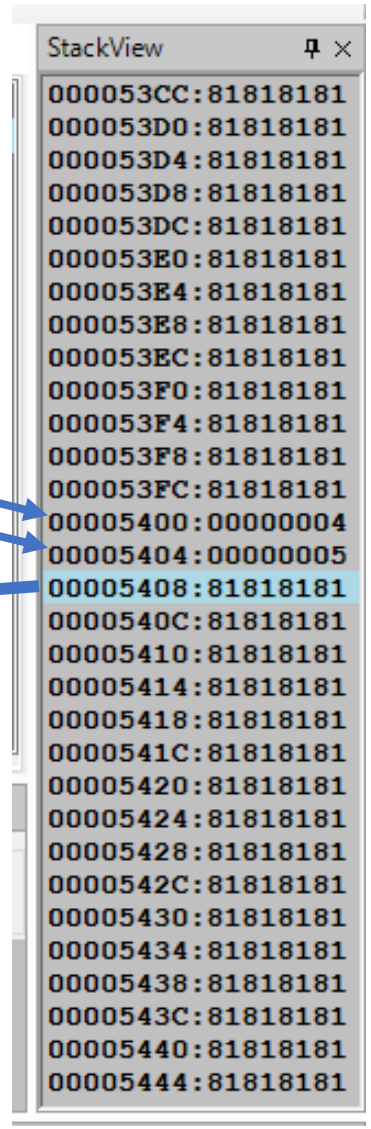
Microprocessor & Computer Architecture (µpCA)

Block Transfer Instructions (Stack)



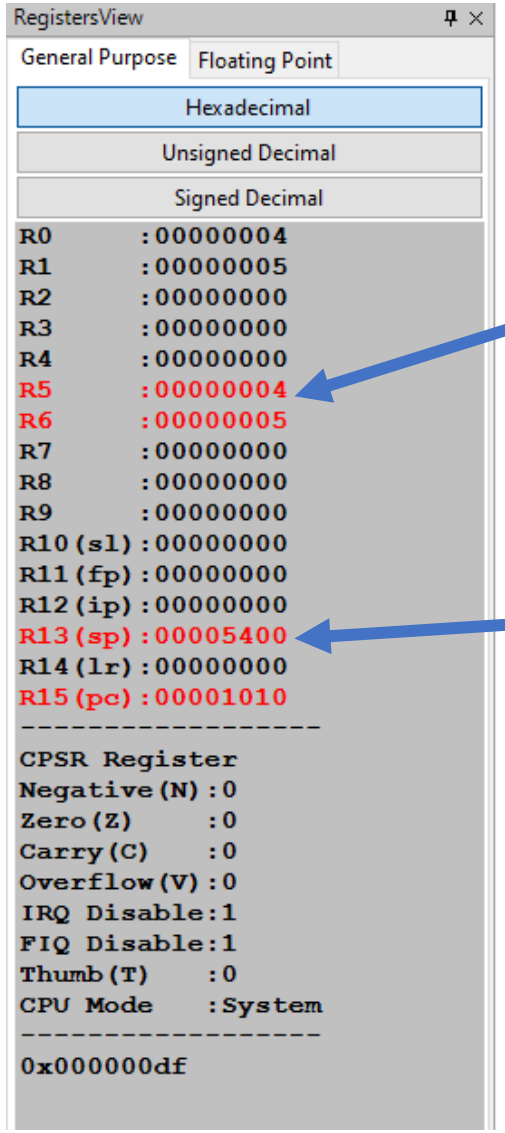

```
.text
MOV R0,#4
MOV R1,#5
STMEA R13!, {R0,R1}
.end
```

PUSH



Microprocessor & Computer Architecture (µpCA)

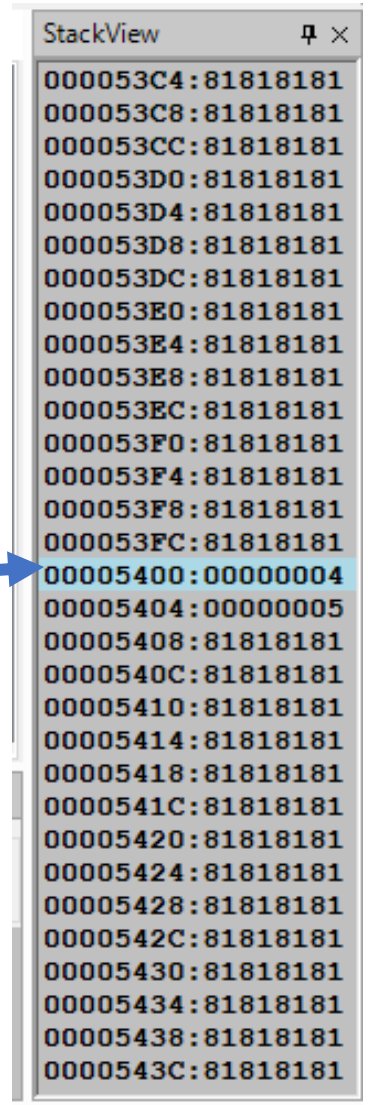
Block Transfer Instructions (Stack)




```
.text
MOV R0,#4
MOV R1,#5
STMEA R13!, {R0,R1}
LDMEA R13!,{R5,R6}
.end
```

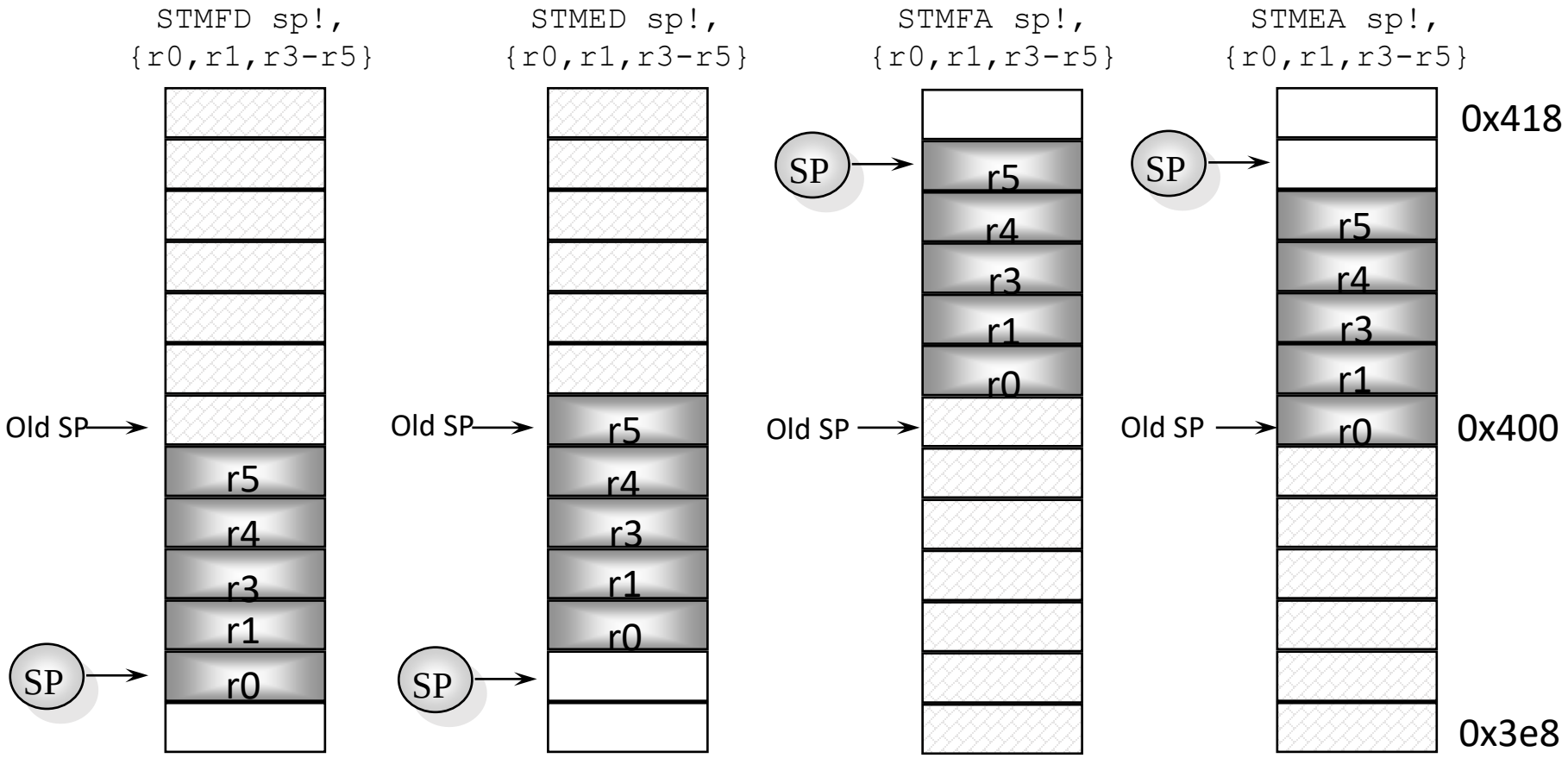
POP

Again the Stack is Empty

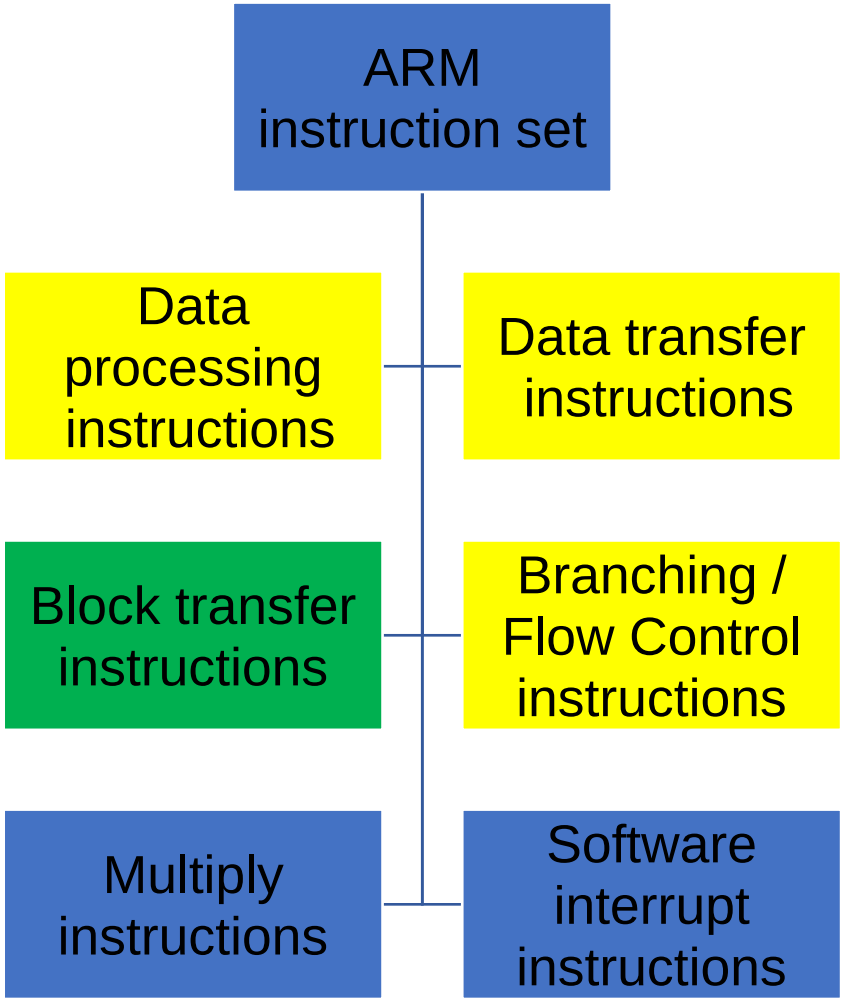


Microprocessor & Computer Architecture (µpCA)

Stack Examples



NEXT Session : Procedure Call



THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

PES UNIVERSITY ONLINE

Microprocessor & Computer Architecture (µpCA)

Block Transfer Instructions

Procedure Call or Subroutine

Dr. D. C. Kiran

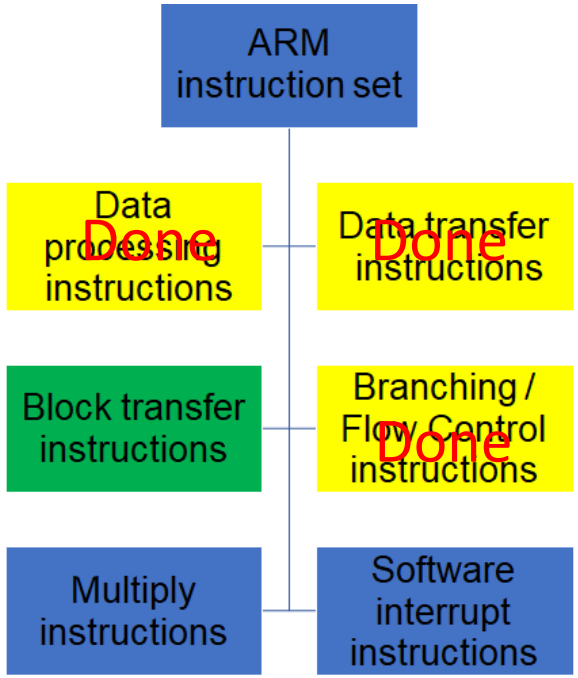
Department of Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Syllabus

Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ~~ARM Program structure~~
- ~~ARM Instruction Format~~
- **ARM INSTRUCTION SET**
  - ~~Data Processing Instructions~~
  - ~~Flow Control Instructions~~
  - ~~Data Transfer Instructions~~
  - ~~Block Transfer Instructions (Stack)~~
  - **Procedure Call**



Procedure Call

R0-R7	
R8-R12	
R13 (SP)	
R14 (LR)	
R15 (PC)	**0x0010**
CPSR	

Main Procedure

0x0000	Instruction 1
0x0004	Instruction 2
0x0008	Instruction 3
0x000C	**Procedure Call**
0x0010	**Instruction 4**
0x0014	Instruction 5
	:::::::::::::::::
	Instruction Last

Called Procedure

0x0020	**Instruction 1**
0x0024	Instruction 2
0x0028	Instruction 3
0xxxxx	Return Instruction

General Structure of Procedure Call:

R0-R7	
R8-R12	
R13 (SP)	
R14 (LR)	**0x0010**
R15 (PC)	**0x0020**
CPSR	

Main Procedure

0x0000	Instruction 1
0x0004	Instruction 2
0x0008	Instruction 3
0x000C	**Procedure Call**
0x0010	**Instruction 4**
0x0014	Instruction 5
	.....
	Instruction Last

Called Procedure

0x0020	**Instruction 1**
0x0024	Instruction 2
0x0028	Instruction 3
0xxxxx	Return Instruction

LR=PC

PC= Address of the 1st Instruction

General Structure of Procedure Return

R0-R7	
R8-R12	
R13 (SP)	
R14 (LR)	
R15 (PC)	**0x0010**
CPSR	

Main Procedure

0x0000	Instruction 1
0x0004	Instruction 2
0x0008	Instruction 3
0x000C	**Procedure Call**
0x0010	**Instruction 4**
0x0014	Instruction 5
	::::::::::::::::::
	Instruction Last

Called Procedure

0x0020	**Instruction 1**
0x0024	Instruction 2
0x0028	Instruction 3
0xxxxx	Return Instruction

PC=LR

Microprocessor & Computer Architecture (µpCA)

General Structure of Procedure Call & Return

Main Procedure

0x0000	Instruction 1
0x0004	Instruction 2
0x0008	Instruction 3
0x000C	**BL Procedure**
0x0010	**Instruction 4**
0x0014	Instruction 5
	:::::::::::::::::
	Instruction Last

Called Procedure

0x0020	**Procedure: Instruction 1**
0x0024	Instruction 2
0x0028	Instruction 3
0xxxxx	**MOV PC LR**
	or
	BX LR

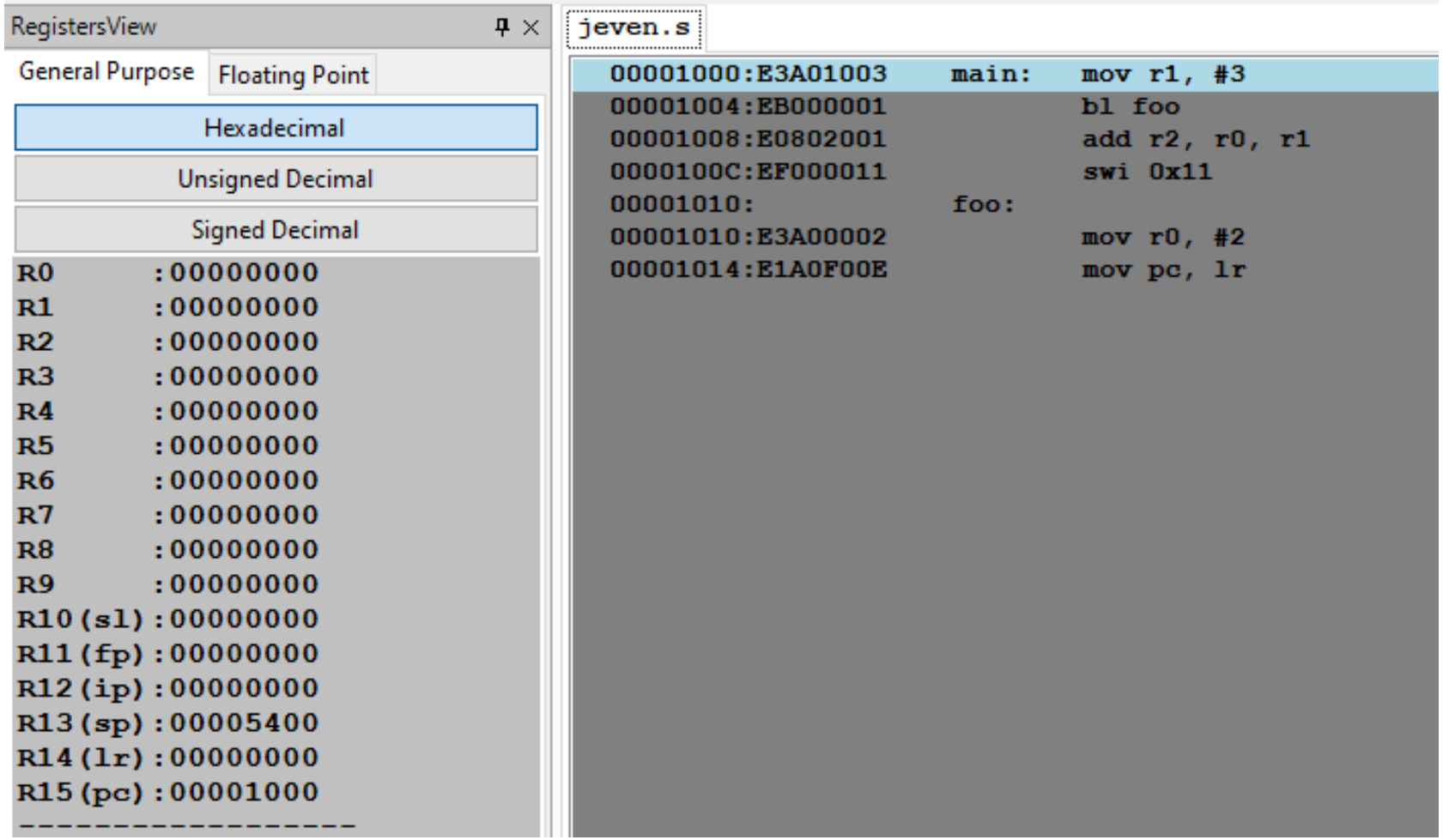
Procedure Call : Example 1

```
main:   mov r1, #3
        bl foo
        add r2, r0, r1
    swi 0x11
foo:
        mov r0, #2
        bx lr
```

```
main:   mov r1, #3
        bl foo
        add r2, r0, r1
    swi 0x11
foo:
        mov r0, #2
        mov pc, lr
```

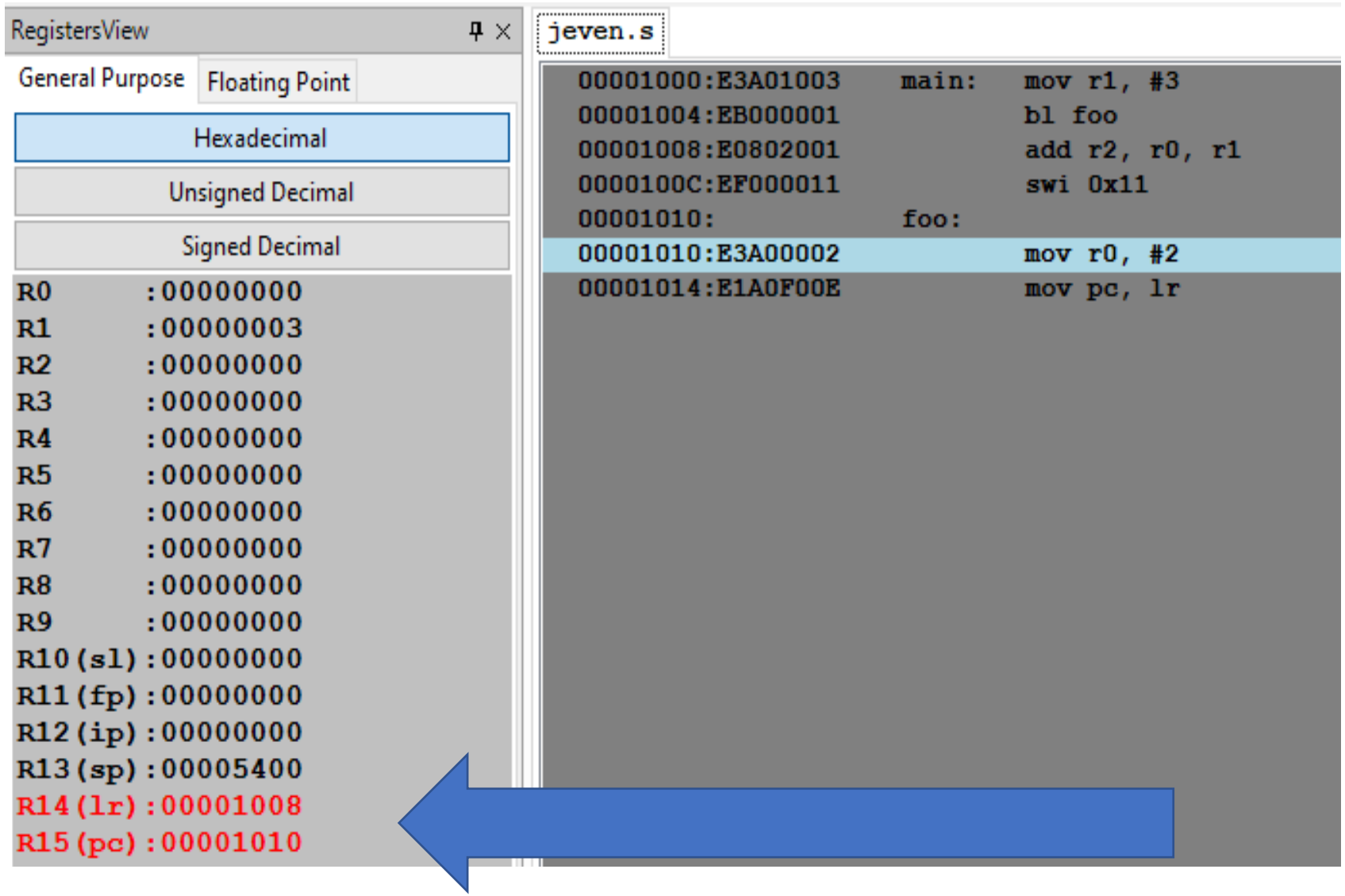
Microprocessor & Computer Architecture (μpCA)

Procedure Call : Example 1



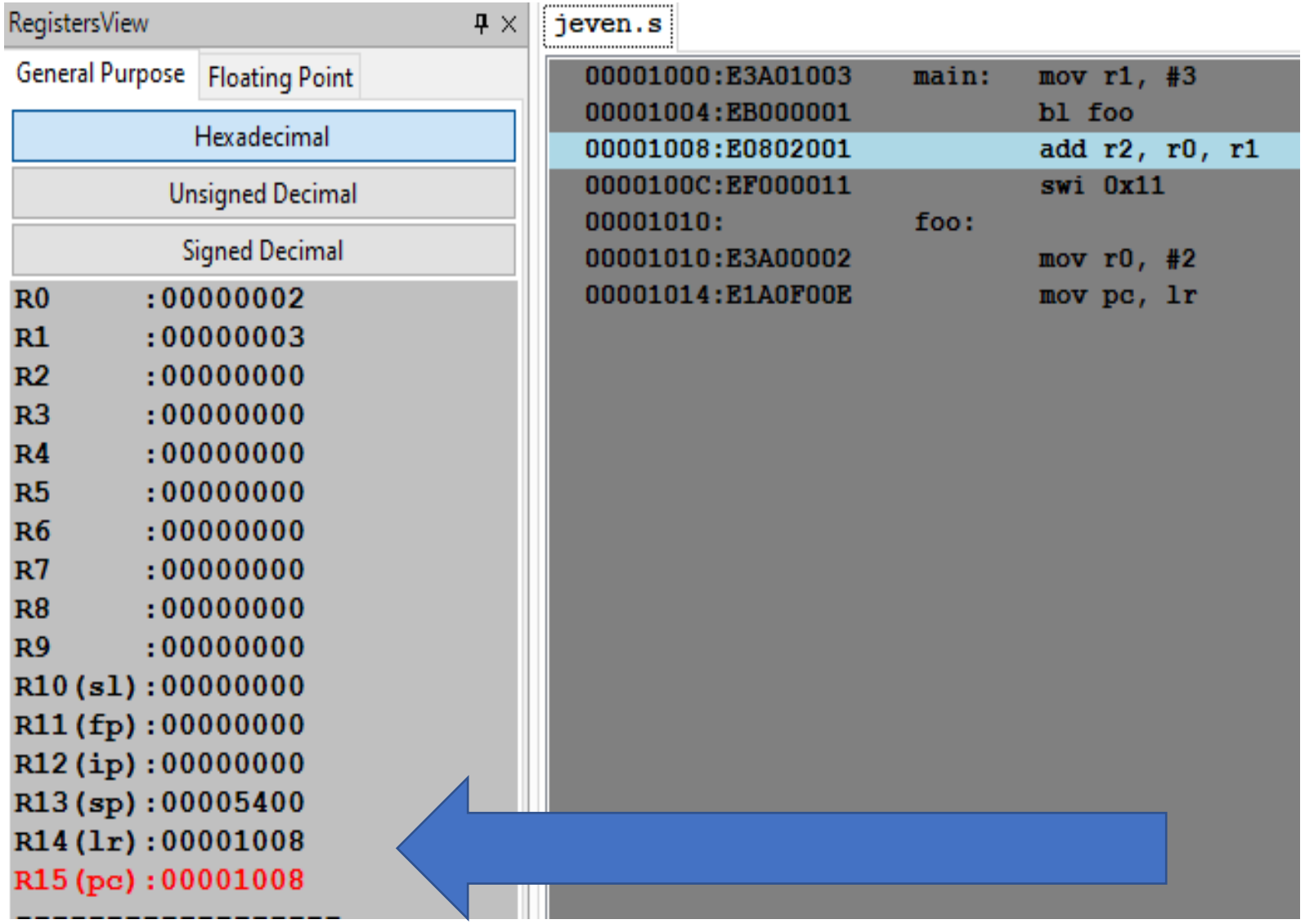
Microprocessor & Computer Architecture (μpCA)

Procedure Call : Example 1



Microprocessor & Computer Architecture (µpCA)

Procedure Call : Example 1

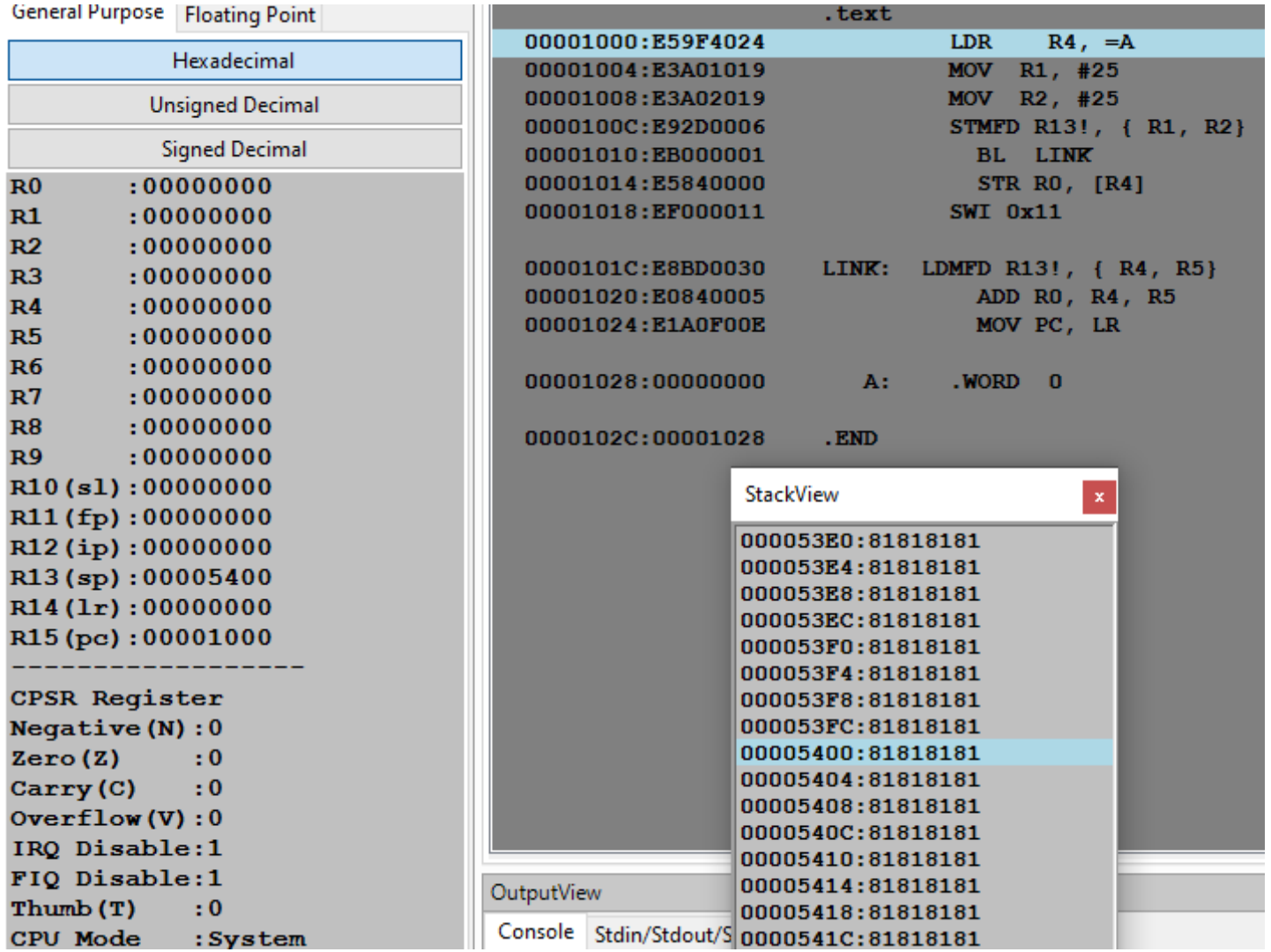


PARAMETER PASSING TO PROCEDURES USING STACK

```
        LDR    R4, =A
        MOV  R1, #25                    ; parameter1
        MOV  R2, #25                    ; parameter2
        STMFD R13!, { R1, R2}           ; parameters are PUSHed on stack.
        BL  LINK
        STR R0, [R4]                    ; return value in Reg. R0.
        SWI 0x11

LINK:  LDMFD R13!, { R4, R5}            ; parameters are POPed from the  stack
        ADD R0, R4, R5                  ;  Result is in register R0.
        MOV PC, LR

  A:    .WORD  0

.END
```

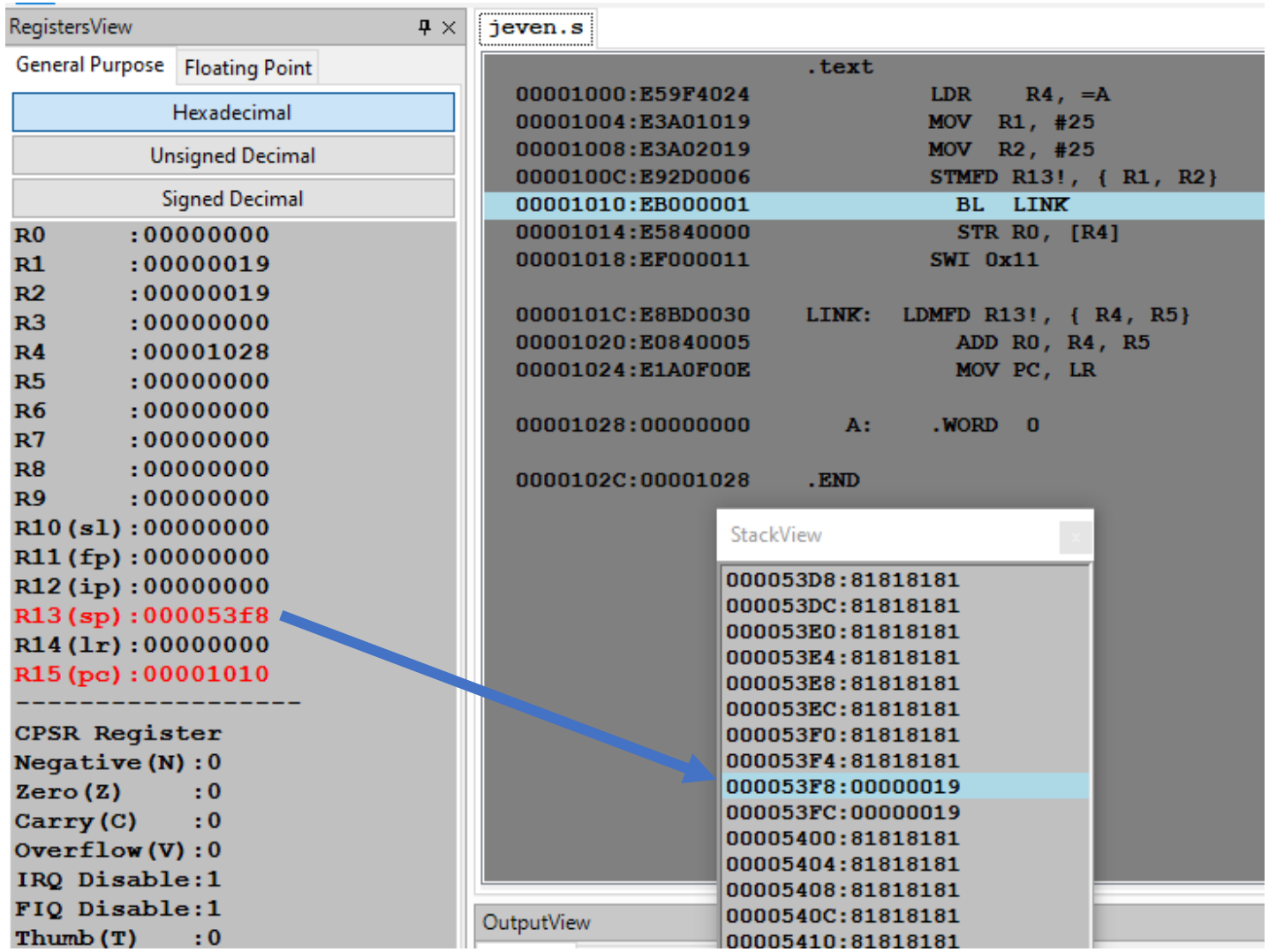
Microprocessor & Computer Architecture (µpCA)

PARAMETER PASSING TO PROCEDURES USING STACK



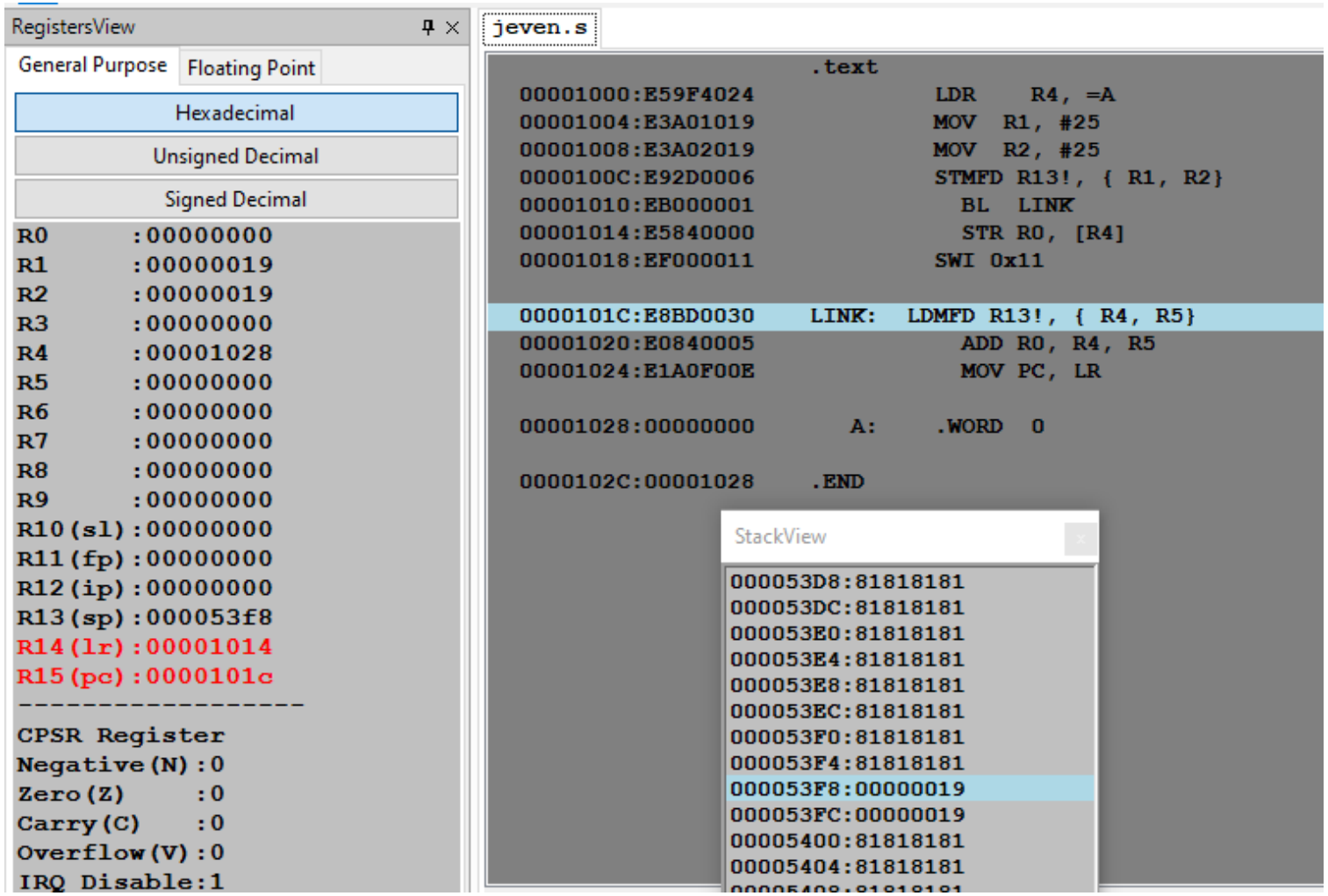
Microprocessor & Computer Architecture (µpCA)

PARAMETER PASSING TO PROCEDURES USING STACK



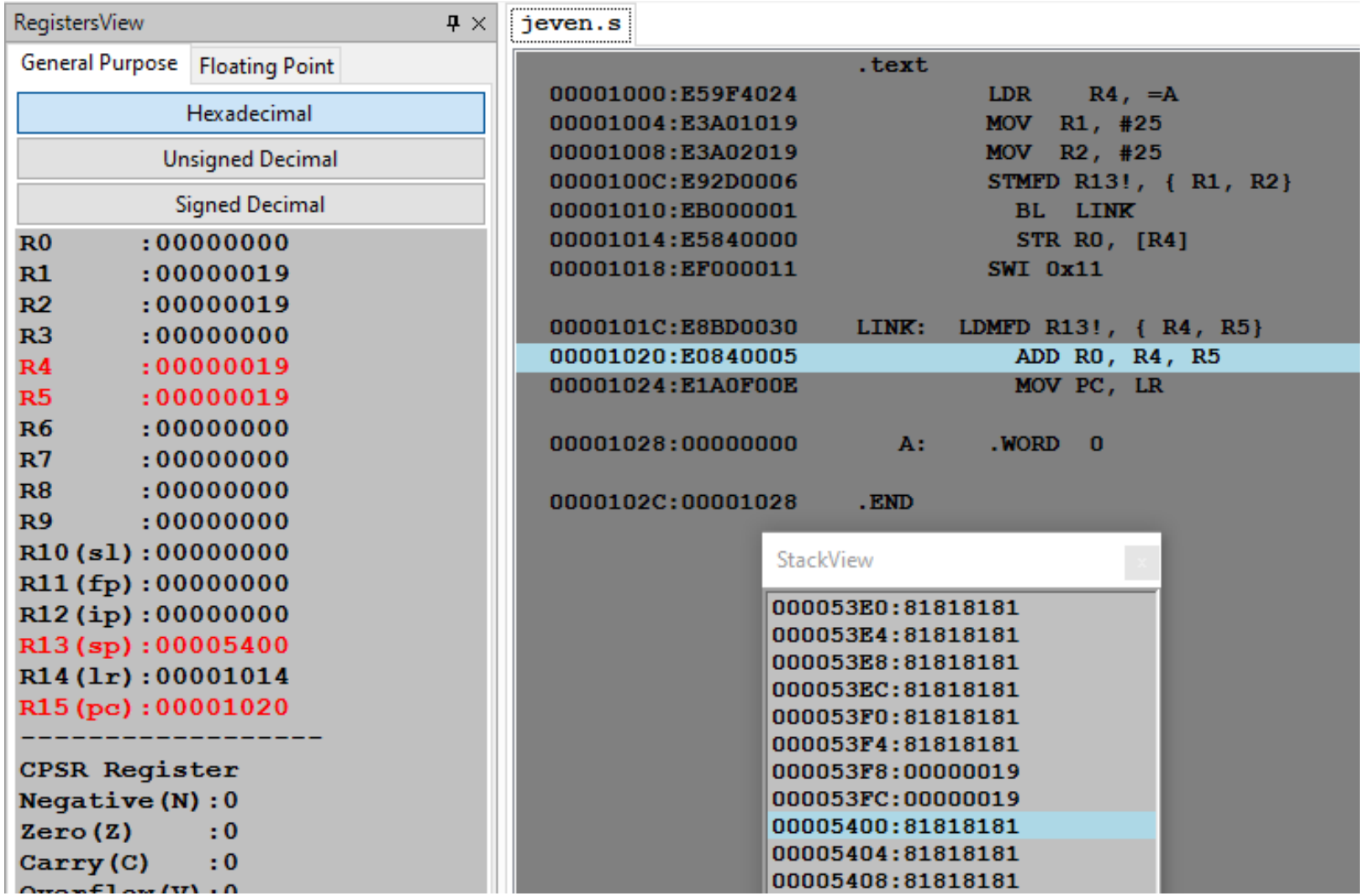
Microprocessor & Computer Architecture (µpCA)

PARAMETER PASSING TO PROCEDURES USING STACK



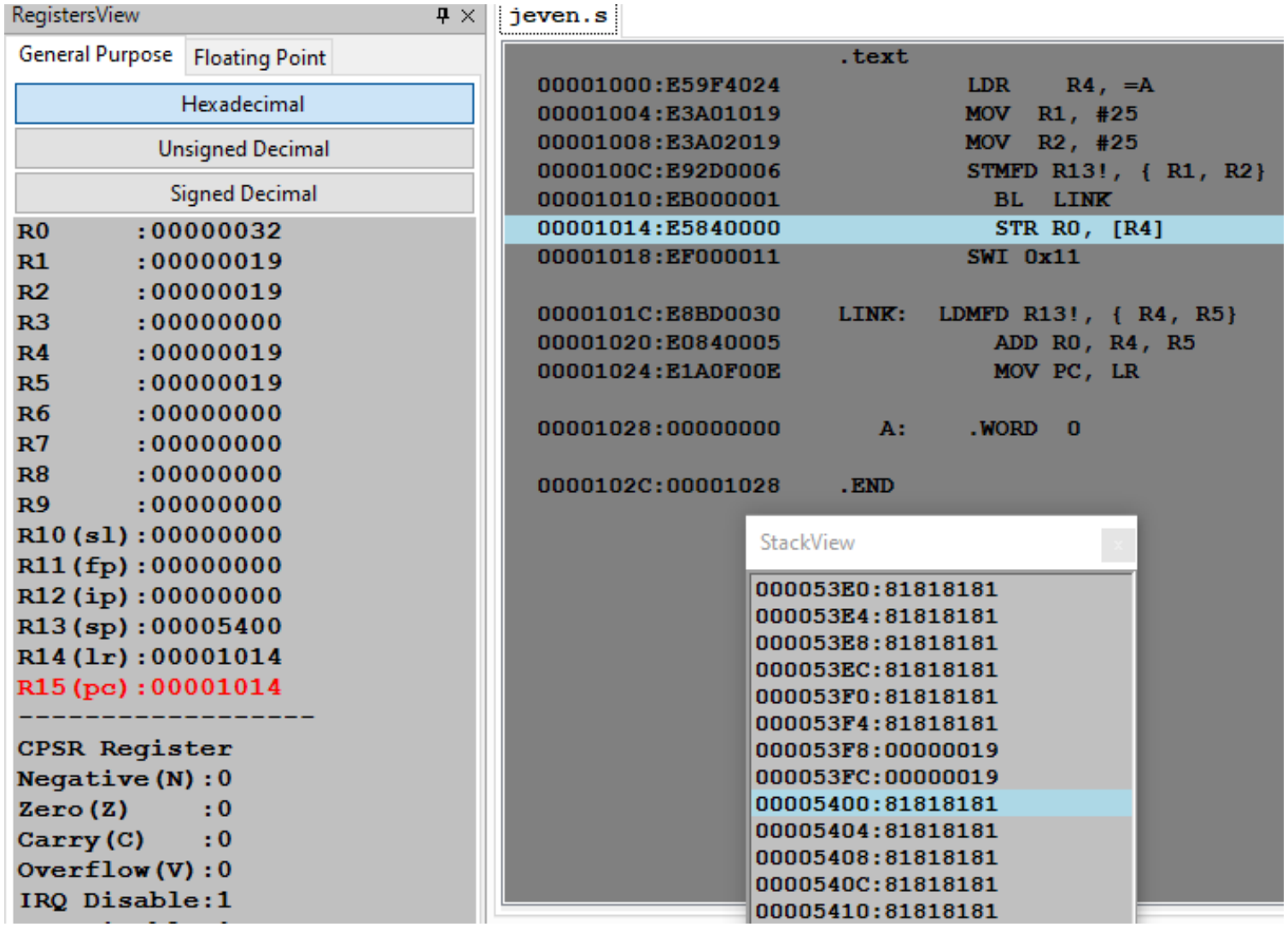
Microprocessor & Computer Architecture (µpCA)

PARAMETER PASSING TO PROCEDURES USING STACK



Microprocessor & Computer Architecture (µpCA)

PARAMETER PASSING TO PROCEDURES USING STACK



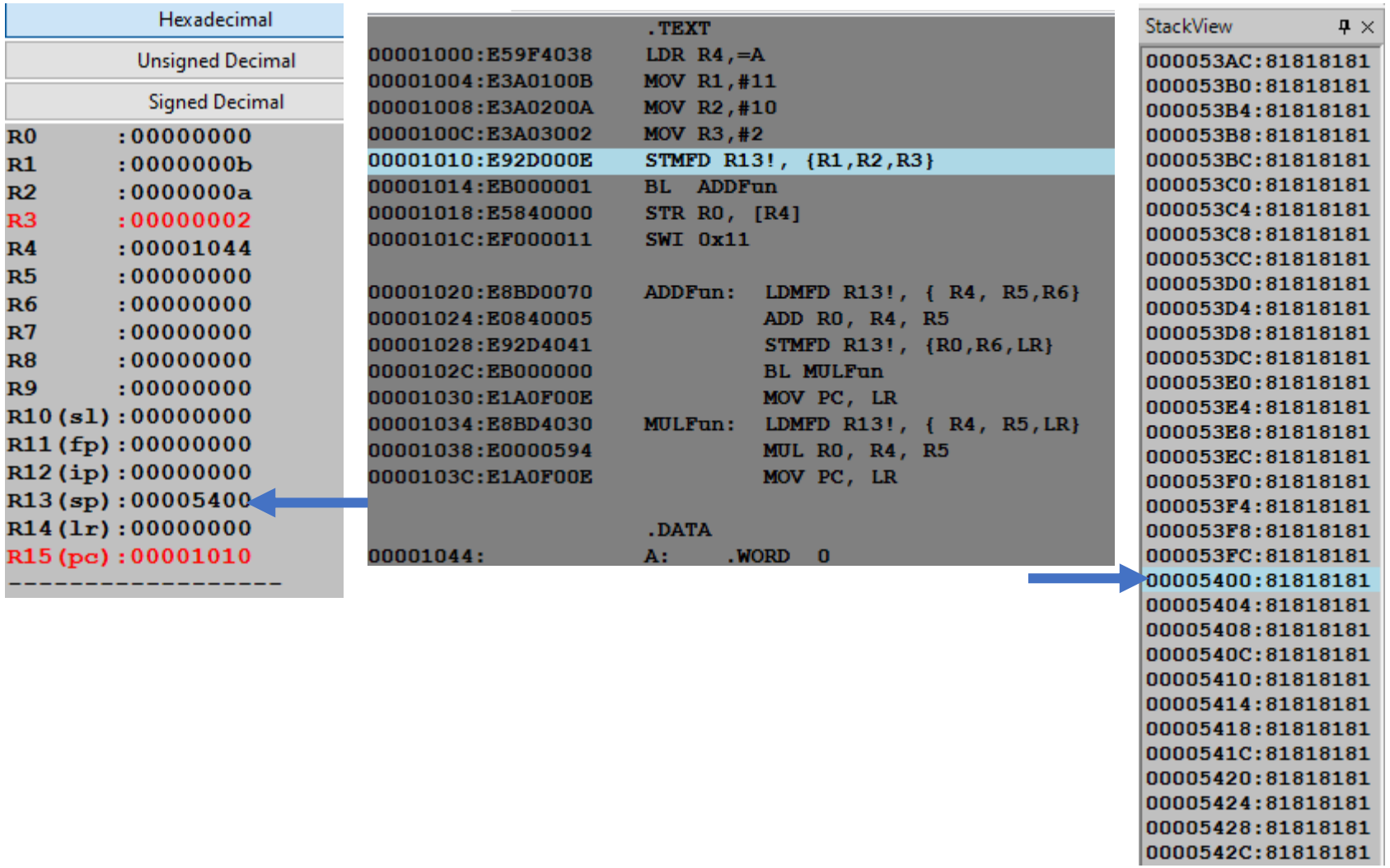
Microprocessor & Computer Architecture (µpCA)

Nested Procedure Call: MUL(ADD(a,b),c)

```
.TEXT       ; MAIN Procedure
LDR R4,=A
MOV R1,#11
MOV R2,#10
MOV R3,#2
STMFD R13!, {R1,R2,R3}
BL  ADDFun                          ; Call to ADD Procedure
STR R0, [R4]
SWI 0x11

ADDFun:  LDMFD R13!, { R4, R5,R6}   ; ADD Procedure
        ADD R0, R4, R5
        STMFD R13!, {R0,R6,LR}
        BL MULFun                   ; Call to MUL Procedure
        MOV PC, LR                  ;Return to Main Procedure
MULFun:  LDMFD R13!, { R4, R5,LR}
        MUL R0, R4, R5
        MOV PC, LR                  ; Return to ADD Procedure

.DATA
A:    .WORD  0
```

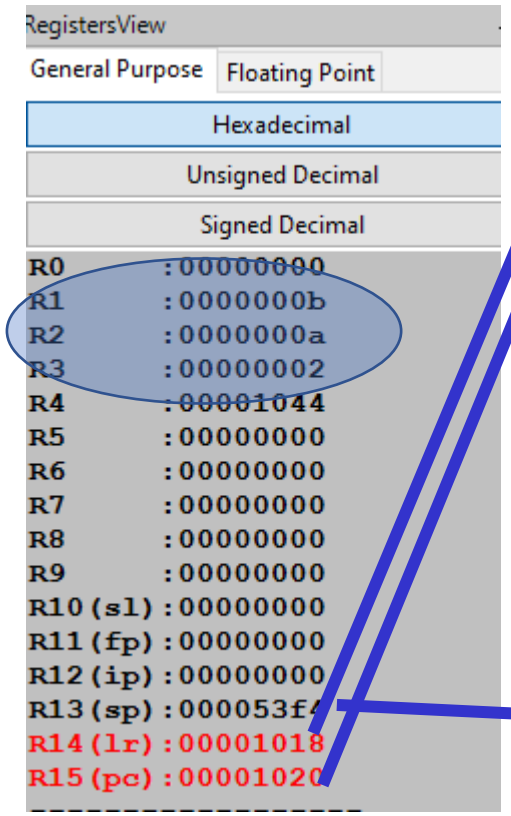
Microprocessor & Computer Architecture (µpCA)

Nested Procedure Call: MUL(ADD(a,b),c)

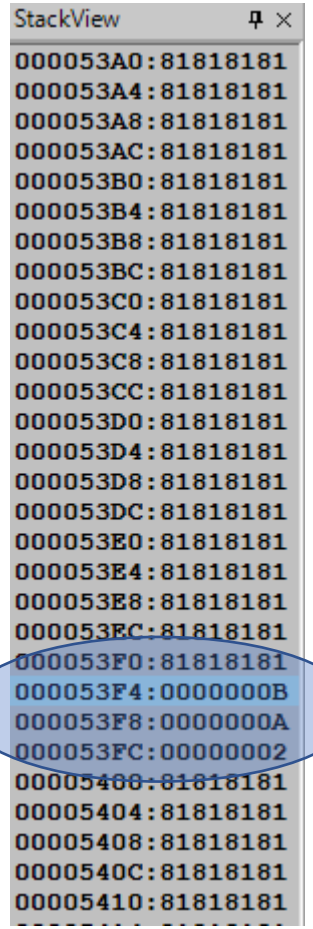


Microprocessor & Computer Architecture (µpCA)

Nested Procedure Call: MUL(ADD(a,b),c)

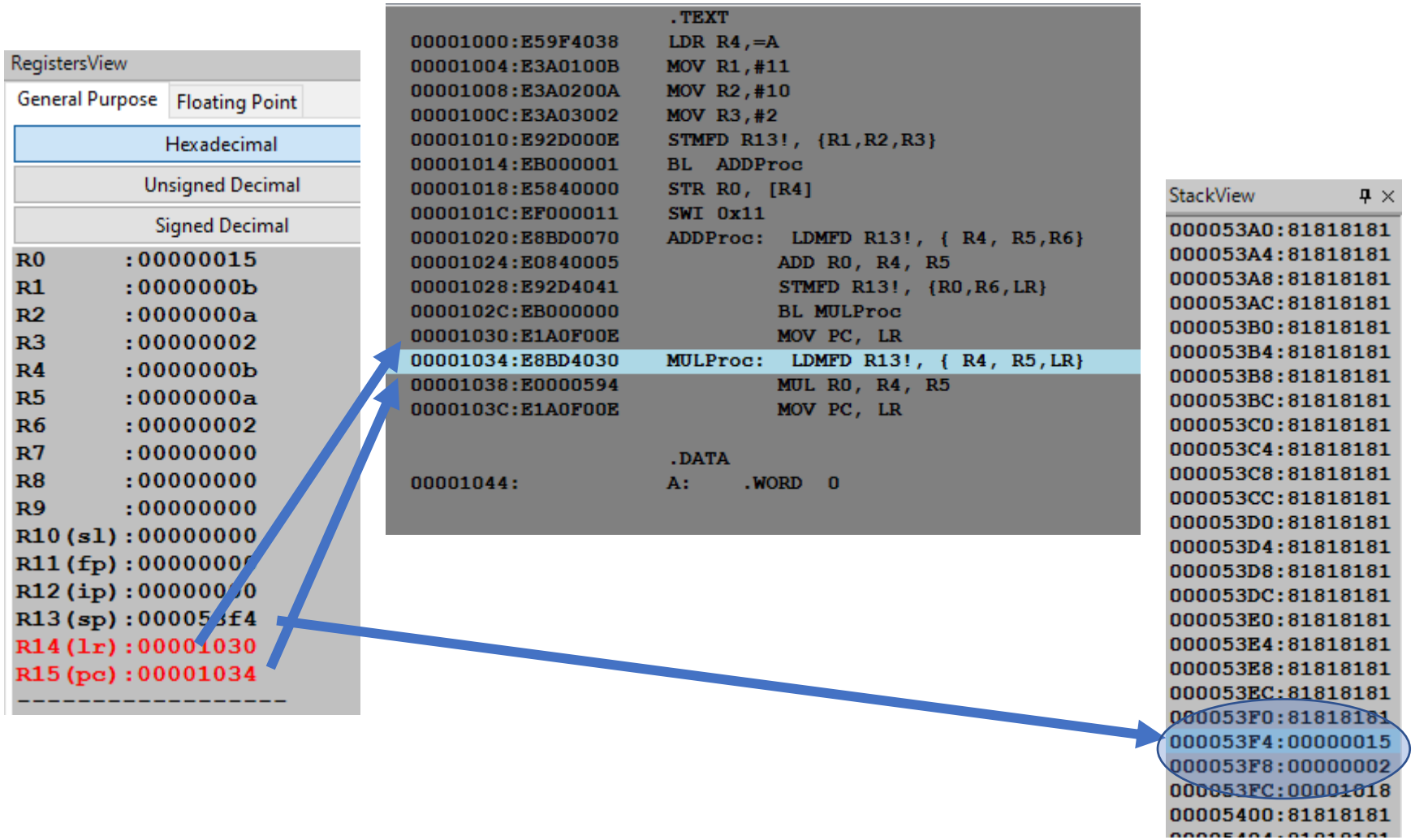



```
                        .TEXT
00001000:E59F4038       LDR R4,=A
00001004:E3A0100B       MOV R1,#11
00001008:E3A0200A       MOV R2,#10
0000100C:E3A03002       MOV R3,#2
00001010:E92D000E       STMFD R13!, {R1,R2,R3}
00001014:EB000001       BL  ADDProc
00001018:E5840000       STR R0, [R4]
0000101C:EF000011       SWI 0x11
00001020:E8BD0070       ADDProc:  LDMFD R13!, { R4, R5,R6}
00001024:E0840005                 ADD R0, R4, R5
00001028:E92D4041                 STMFD R13!, {R0,R6,LR}
0000102C:EB000000                 BL MULProc
00001030:E1A0F00E                 MOV PC, LR
00001034:E8BD4030       MULProc:  LDMFD R13!, { R4, R5,LR}
00001038:E0000594                 MUL R0, R4, R5
0000103C:E1A0F00E                 MOV PC, LR

                        .DATA
00001044:               A:    .WORD  0
```



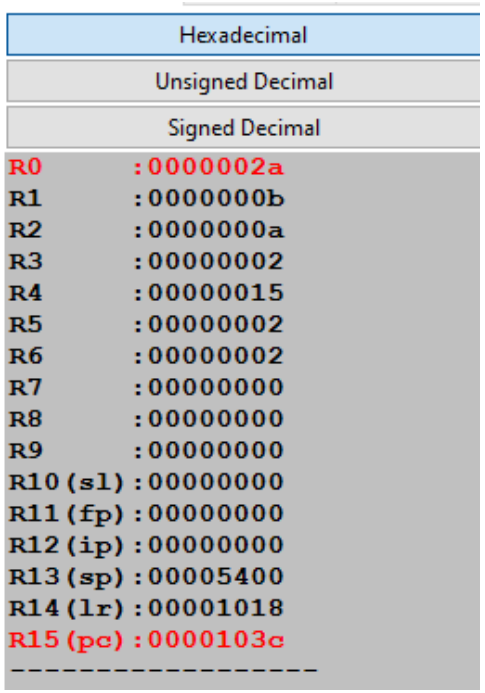
Microprocessor & Computer Architecture (µpCA)

Nested Procedure Call: MUL(ADD(a,b),c)



Microprocessor & Computer Architecture (µpCA)

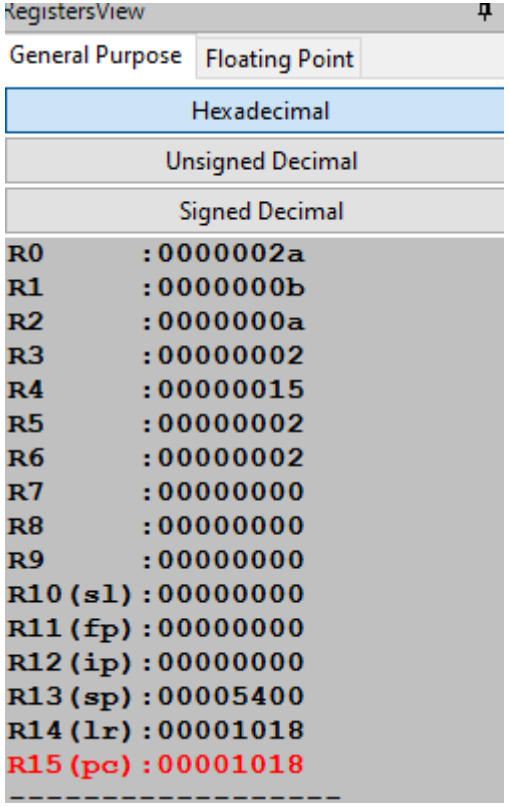
Nested Procedure Call: MUL(ADD(a,b),c)




```
                        .TEXT
00001000:E59F4038       LDR R4,=A
00001004:E3A0100B       MOV R1,#11
00001008:E3A0200A       MOV R2,#10
0000100C:E3A03002       MOV R3,#2
00001010:E92D000E       STMFD R13!, {R1,R2,R3}
00001014:EB000001       BL  ADDProc
00001018:E5840000       STR R0, [R4]
0000101C:EF000011       SWI 0x11
00001020:E8BD0070       ADDProc:  LDMFD R13!, { R4, R5,R6}
00001024:E0840005                 ADD R0, R4, R5
00001028:E92D4041                 STMFD R13!, {R0,R6,LR}
0000102C:EB000000                 BL MULProc
00001030:E1A0F00E                 MOV PC, LR
00001034:E8BD4030       MULProc:  LDMFD R13!, { R4, R5,LR}
00001038:E0000594                 MUL R0, R4, R5
0000103C:E1A0F00E                 MOV PC, LR

                        .DATA
00001044:               A:     .WORD  0
```

Microprocessor & Computer Architecture (µpCA)

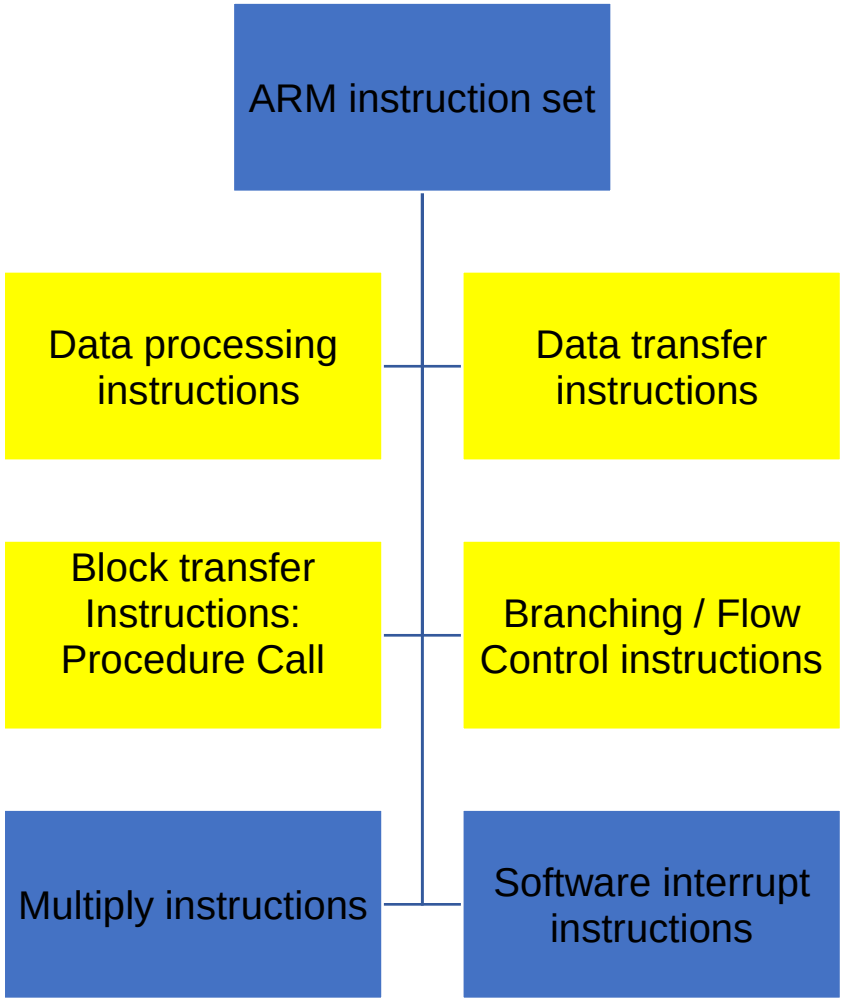
Nested Procedure Call: MUL(ADD(a,b),c)




```
                        .TEXT
00001000:E59F4038       LDR R4,=A
00001004:E3A0100B       MOV R1,#11
00001008:E3A0200A       MOV R2,#10
0000100C:E3A03002       MOV R3,#2
00001010:E92D000E       STMFD R13!, {R1,R2,R3}
00001014:EB000001       BL  ADDProc
00001018:E5840000       STR R0, [R4]
0000101C:EF000011       SWI 0x11
00001020:E8BD0070       ADDProc:  LDMFD R13!, { R4, R5,R6}
00001024:E0840005                 ADD R0, R4, R5
00001028:E92D4041                 STMFD R13!, {R0,R6,LR}
0000102C:EB000000                 BL MULProc
00001030:E1A0F00E                 MOV PC, LR
00001034:E8BD4030       MULProc:  LDMFD R13!, { R4, R5,LR}
00001038:E0000594                 MUL R0, R4, R5
0000103C:E1A0F00E                 MOV PC, LR

                        .DATA
00001044:               A:    .WORD  0
```

Microprocessor & Computer Architecture (µpCA)

NEXT Session : Multiplication



THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

PES UNIVERSITY ONLINE

Microprocessor & Computer Architecture (µpCA)

Multiple Register Load / Store
or
Block Transfer Instructions

Dr. D. C. Kiran
Department of Computer Science and Engineering

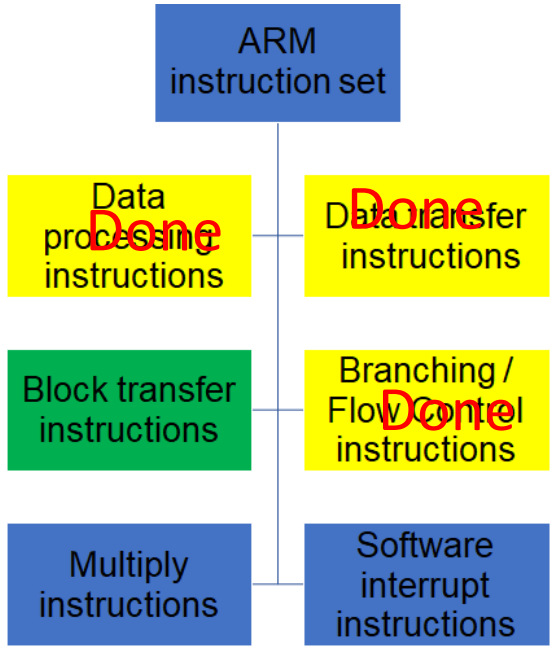
Microprocessor & Computer Architecture (µpCA)

Syllabus

Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ~~ARM Program structure~~
- ~~ARM Instruction Format~~
- **ARM INSTRUCTION SET**
  - ~~Data Processing Instructions~~
  - ~~Flow Control Instructions~~
  - ~~Data Transfer Instructions~~

Multiple Register Load / Store (Block Transfer Instructions)



Single Register Load and Store

Consider a Scenario where you need to transfer, 10,20,30 to register R1,R2,R3

```
.text
LDR R4,=A
LDR R1,[R4],#4
LDR R2,[R4],#4
LDR R3,[R4],#4
.data
A:.word 10,20,30,40,50
```

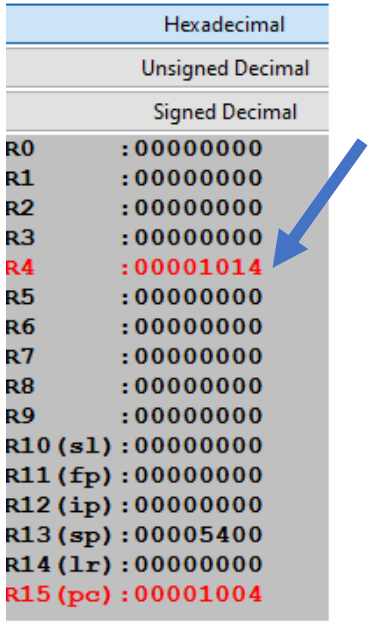
Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

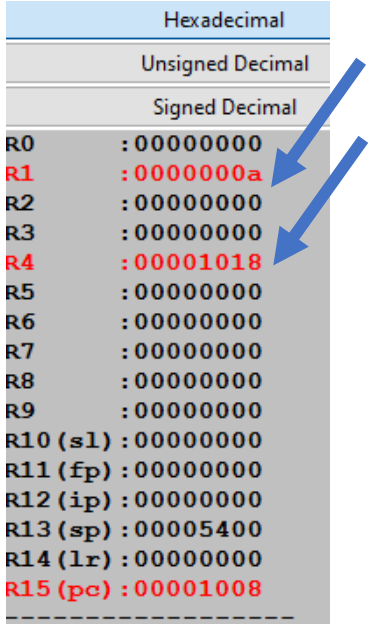
Microprocessor & Computer Architecture (µpCA)

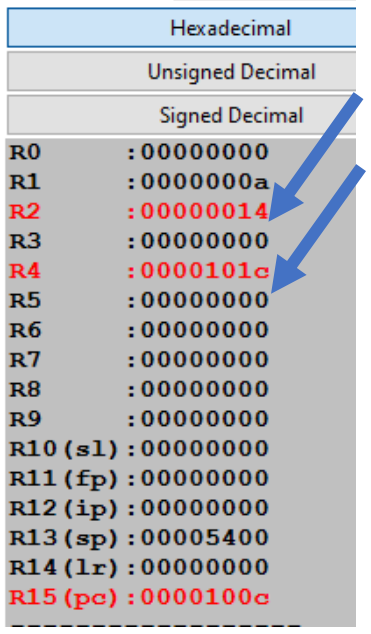
Single Register Load and Store

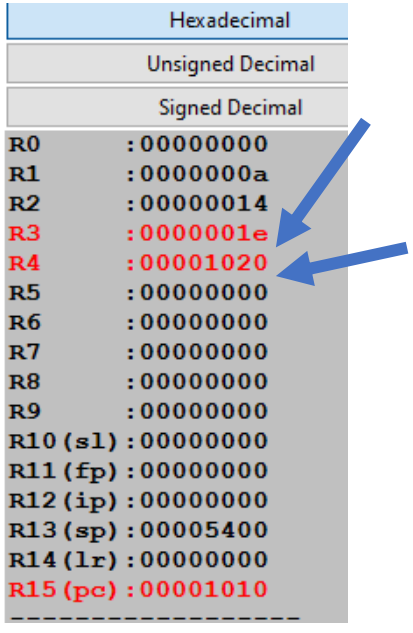
```
                .text
                 LDR R4,=A
                 LDR R1,[R4],#4
                 LDR R2,[R4],#4
                 LDR R3,[R4],#4
                .data
00001014        A:.word 10,20,30,40,50
```

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60









Microprocessor & Computer Architecture (µpCA)

Single Register Load and Store

Hexadecimal
Unsigned Decimal
Signed Decimal

```
R0      :00000000
R1      :0000000a
R2      :00000014
R3      :0000001e
R4      :00001048
R5      :00000000
R6      :00000000
R7      :00000000
R8      :00000000
R9      :00000000
R10(sl) :00000000
R11(fp) :00000000
R12(ip) :00000000
R13(sp) :00005400
R14(lr) :00000000
R15(pc) :00011400
```

```
.text
LDR R4,=A
LDR R1,[R4],#4
LDR R2,[R4],#4
LDR R3,[R4],#4
LDR R4,=B
STR R1,[R4],#4
STR R2,[R4],#4
STR R3,[R4],#4
.data
A:.word 10,20,30,40,50
000103C  B:.word
```

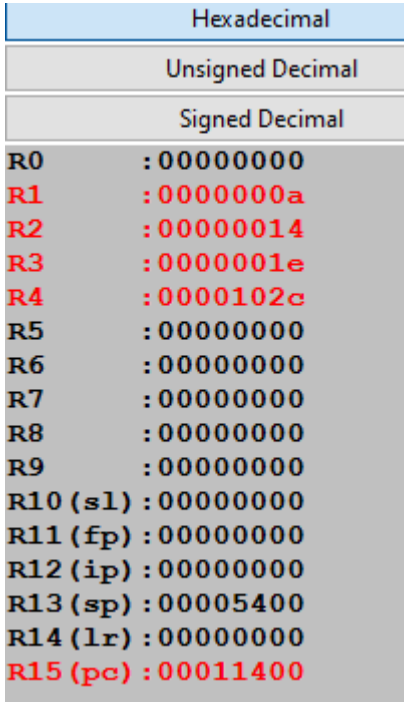
MemoryView1

0000103c

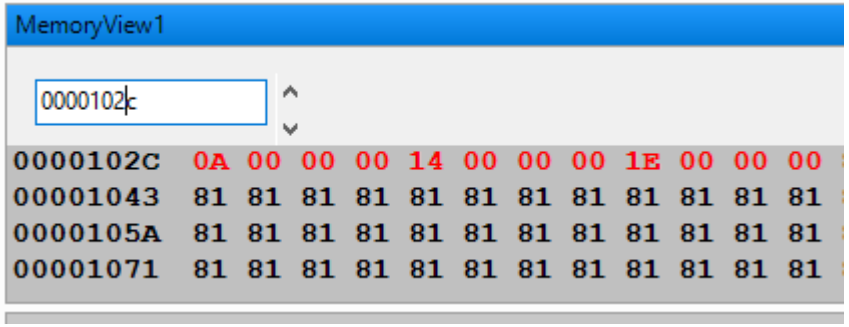
```
0000103C  0A 00 00 00 14 00 00 00 1E 00 00 00
00001053  81 81 81 81 81 81 81 81 81 81 81 81
0000106A  81 81 81 81 81 81 81 81 81 81 81 81
00001081  81 81 81 81 81 81 81 81 81 81 81 81
```

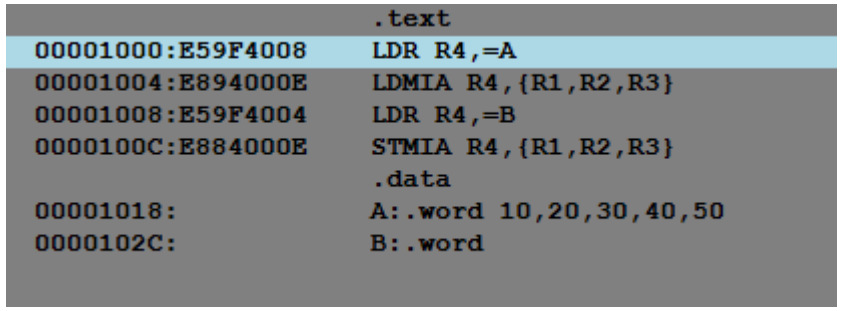
Microprocessor & Computer Architecture (µpCA)

Multiple Register Load and Store




```
.text
LDR R4,=A
LDMIA R4,{R1,R2,R3}
LDR R4,=B
STMIA R4,{R1,R2,R3}
.data
A:.word 10,20,30,40,50
000102C B:.word
```





Multiple Register Load and Store or Block Transfer

LDM: **Load multiple registers**

STM: **Store multiple registers**

Syntax:

<LDM/STM> {cond} <Addressing Mode>Rn {!},Registers

Multiple Register Load and Store or Block Transfer

Specifying Addressing Mode

Syntax:

<LDM/STM> {cond} <Addressing Mode>Rn {!},Registers

Addressing Mode	Meaning
IA	Increase after
IB	Increase before
DA	Decrease after
DB	Decrease before

Specifying Registers

Syntax:

<LDM/STM> {cond} <Addressing Mode>Rn {!},Registers

LDM <IA/IB/DA/DB> Rn, {R1,R2,R3}

or

LDMIA <IA/IB/DA/DB> Rn, {R1-R3}

LDMIA

LDM**IA** Rn, {Ri,Ri+1,Ri+2….}
Let Rn=0x101C

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMIA R0, {R1,R2,R3}
or
LDMIA R0, {R1-R3}

R1=30
R2=40
R3=50

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMIA

LDM**IA** Rn**!**, {Ri,Ri+1,Ri+2….}
Initially R0= 0x101C

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMIA R0!, {R1,R2,R3}
or
LDMIA R0!, {R1-R3}

R1=?
R2=?
R3=?

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMIA

LDM**IA** Rn**!**, {Ri,Ri+1,Ri+2….}
Initially R0= 0x101C

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMIA R0!, {R1,R2,R3}
or
LDMIA R0!, {R1-R3}

R1=30 Copy First
R2=?
R3=?

Increment After
R0 →

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMIA

LDM**IA** Rn**!**, {Ri,Ri+1,Ri+2….}
Initially R0= 0x101C

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMIA R0!, {R1,R2,R3}
or
LDMIA R0!, {R1-R3}

R1=30
R2=40
R3=?

Copy First

Increment After

R0 →

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMIA

LDM**IA** Rn**!**, {Ri,Ri+1,Ri+2….}
Initially R0= 0x101C

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMIA R0!, {R1,R2,R3}
or
LDMIA R0!, {R1-R3}

R1=30
R2=40
R3=50

Copy First

Increment After

R0 →

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMIB

LDM**IB** Rn**!**, {Ri,Ri+1,Ri+2….}

Initially Rn= 0x101C

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMIB R0!, {R1,R2,R3}
or
LDMIB R0!, {R1-R3}

R1=?
R2=?
R3=?

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMIB

LDM**IB** Rn**!**, {Ri,Ri+1,Ri+2….}

Initially Rn= 0x101C

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMIB R0!, {R1,R2,R3}
or
LDMIB R0!, {R1-R3}

R1=40 Copy After
R2=?
R3=?

Increment First
R0 →

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMIB

LDM**IB** Rn**!**, {Ri,Ri+1,Ri+2….}

Initially Rn= 0x101C

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMIB R0!, {R1,R2,R3}
or
LDMIB R0!, {R1-R3}

R1=40
R2=50 Copy After
R3=?

Increment First
R0 →

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMIB

LDM**IB** Rn**!**, {Ri,Ri+1,Ri+2….}

Initially Rn= 0x101C

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMIB R0!, {R1,R2,R3}
or
LDMIB R0!, {R1-R3}

R1=40
R2=50
R3=60

Copy After

Increment First
R0 →

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMDA

LDM**DA** Rn**!**, {Ri,Ri+1,Ri+2….}

Initially Rn=0x1010

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMDA R0!, {R1,R2,R3}
or
LDMDA R0!, {R1-R3}

R1=?
R2=?
R3=?

R0 →

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMDA

LDM**DA** Rn**!**, {Ri,Ri+1,Ri+2….}

Initially Rn=0x1010

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMDA R0!, {R1,R2,R3}
or
LDMDA R0!, {R1-R3}

R1=40 Copy First
R2=
R3=

Decrement After
R0 →

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMDA

LDM**DA** Rn**!**, {Ri,Ri+1,Ri+2….}

Initially Rn=0x1010

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMDA R0!, {R1,R2,R3}
or
LDMDA R0!, {R1-R3}

R1=40
R2=30
R3=

Copy First

Decrement After
R0 →

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMDA

LDM**DA** Rn**!**, {Ri,Ri+1,Ri+2….}

Initially Rn=0x1010

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMDA R0!, {R1,R2,R3}
or
LDMDA R0!, {R1-R3}

R1=40
R2=30
R3=20 Copy First

Decrement After
R0 →

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMDB

LDM**DB** Rn**!**, {Ri,Ri+1,Ri+2….}

Initially Rn=0x1010

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMDB R0!, {R1,R2,R3}
or
LDMDB R0!, {R1-R3}

R1=?
R2=?
R3=?

R0 →

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMDB

LDM**DB** Rn**!**, {Ri,Ri+1,Ri+2….}

Initially Rn=0x1010

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMDB R0!, {R1,R2,R3}
or
LDMDB R0!, {R1-R3}

R1=30 Copy After
R2=
R3=

Decrement First
R0 →

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMDB

LDM**DB** Rn**!**, {Ri,Ri+1,Ri+2….}

Initially Rn=0x1010

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMDB R0!, {R1,R2,R3}
or
LDMDB R0!, {R1-R3}

R1=30
R2=20
R3=

Copy After

Decrement First
R0 →

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

LDMDB

LDM**DB** Rn**!**, {Ri,Ri+1,Ri+2….}

Initially Rn=0x1010

Ri =[Rn]
Ri+1=[Rn+4]
Ri+2=[Rn+8]
……

Example:

LDMDB R0!, {R1,R2,R3}
or
LDMDB R0!, {R1-R3}

R1=30
R2=20
R3=10

Copy After

Decrement First
R0 →

Addr	data
0x1014	10
0x1018	20
0x101C	30
0x1020	40
0x1024	50
0x1028	60

Think and Relate

IA: addr:=Rn

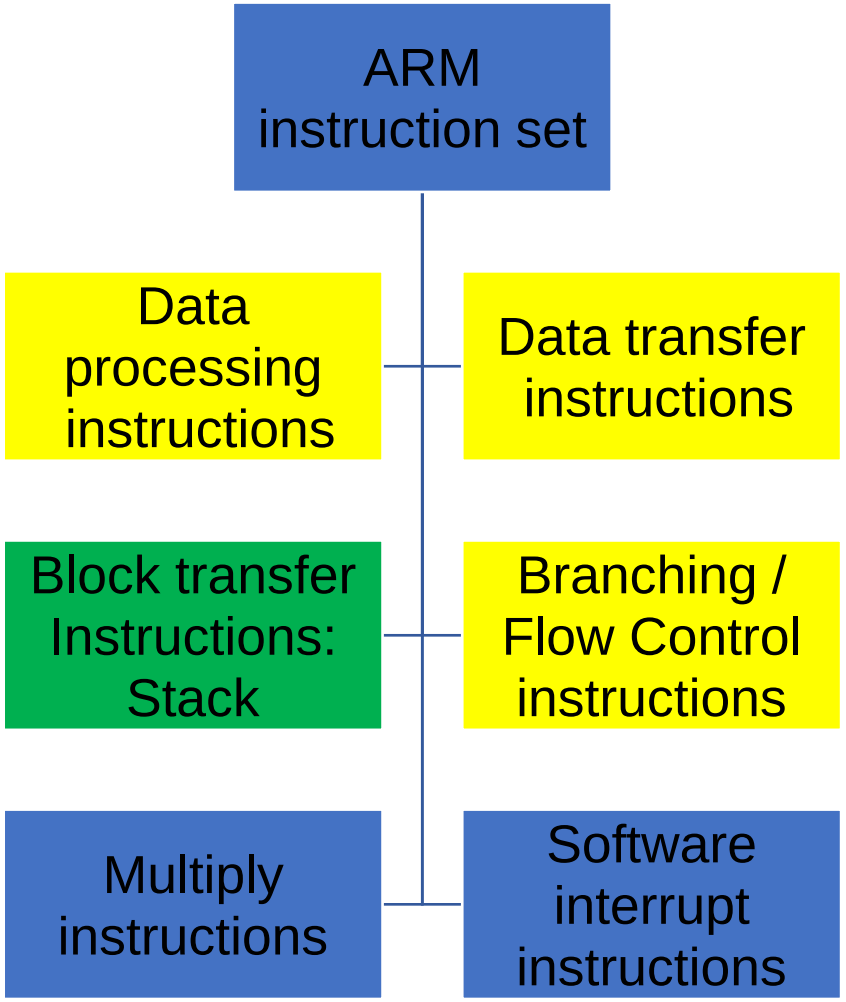
IB: addr:=Rn+4

DA: addr:=Rn-#<registers>*4+4

DB: addr:=Rn-#<registers>*4

Addressing mode	Description	Start address	End address	*Rn*!
IA	increment after	Rn	$Rn + 4^*N - 4$	$Rn + 4^*N$
IB	increment before	$Rn + 4$	$Rn + 4^*N$	$Rn + 4^*N$
DA	decrement after	$Rn - 4^*N + 4$	Rn	$Rn - 4^*N$
DB	decrement before	$Rn - 4^*N$	$Rn - 4$	$Rn - 4^*N$

Next Session



THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Multiplication, PSR & SWAP Instructions

Dr. D. C. Kiran
Department of Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Syllabus

Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ~~ARM Program structure~~
- ~~ARM Instruction Format~~
- **ARM INSTRUCTION SET**
  - ~~Data Processing Instructions~~
  - ~~Flow Control Instructions~~
  - ~~Data Transfer Instructions~~
  - ~~Block Transfer Instructions (Stack & Procedure Call)~~
  - Multiplication
  - MSR & MRS Instructions
  - Swap

Multiplication

MUL	**Multiply**	**32-bit result**
MLA	**Multiply accumulate**	**32-bit result**
UMULL	**Unsigned multiply**	**64-bit result**
UMLAL	**Unsigned multiply accumulate**	**64-bit result**
SMULL	**Signed multiply**	**64-bit result**
SMLAL	**Signed multiply accumulate**	**64-bit result**

Multiplication

MUL{<cond>}{S}Rd,Rf,Rs ;Rd=(RfxRs)$_{[31:0]}$

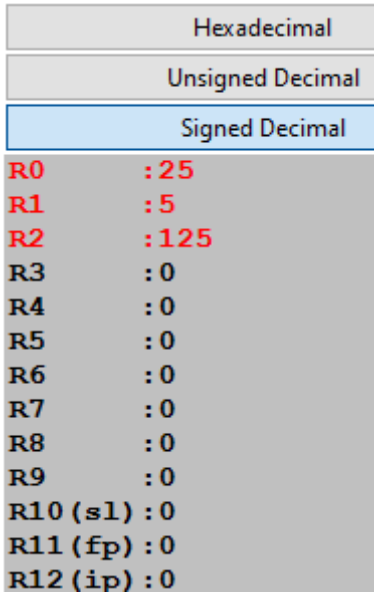
- **MUL R0, R1, R2 @ R0 = (R1xR2)$_{[31:0]}$**

- Features:
  - **Second operand** can't be immediate
  - The result register must be different from the first operand

    R**d**≠R**f**

Microprocessor & Computer Architecture (µpCA)

Multiplication




```
.text
 MOV R0,#25
MOV R1,#5
MUL R2,R0,R1
 .end
```

Microprocessor & Computer Architecture (µpCA)

Multiplication c=c+a[i]*b[i]

```
                        .TEXT
00001000:E59F5030       LDR R5,=A
00001004:E59F6030       LDR R6,=B
00001008:E59F7030       LDR R7,=C
0000100C:E3A03003       Mov R3,#3     ;count Of Numbers
00001010:E3A04000       MOV R4,#0
00001014:               Loop:
00001014:E4951004               Ldr R1,[R5],#4
00001018:E4962004               Ldr R2,[R6],#4
0000101C:E0000291               Mul R0,R1,R2     ;multiple R1 And R2 And Store Result In R0
00001020:E0844000               Add R4,R4,R0
00001024:E4874004               STR R4,[R7],#4
00001028:E2433001               Sub R3,R3,#1
0000102C:E3330000               Teq R3,#0        ; Test For Equality
00001030:1AFFFFF7               Bne Loop
00001034:EF000011               Swi 0x11       ; Interrupt For Termination Of Program

                        .DATA
00001044:               A: .word 1,3,5
00001050:               B: .word 2,4,6
0000105C:               C: .word
```

Microprocessor & Computer Architecture (µpCA)

Multiplication c=c+a[i]*b[i]

R0	:00000000
R1	:00000000
R2	:00000000
R3	:00000005
R4	:00000000
R5	:00001044
R6	:00001050
R7	:0000105c
R8	:00000000

R4=0

R0	:00000002
R1	:00000001
R2	:00000002
R3	:00000005
R4	:00000002
R5	:00001048
R6	:00001054
R7	:0000105c
R8	:00000000

=2

R4=0+2

R0	:0000000c
R1	:00000003
R2	:00000004
R3	:00000004
R4	:0000000e
R5	:0000104c
R6	:00001058
R7	:00001060
R8	:00000000

=14

R4=2+12

R0	:0000001e
R1	:00000005
R2	:00000006
R3	:00000003
R4	:0000002c
R5	:00001050
R6	:0000105c
R7	:00001064
R8	:00000000

=44

R4=14+30

MemoryView1

0000105C

```
0000105C  0002 0000 000E 0000 002C 0000
00001080  8181 8181 8181 8181 8181 8181
```

Multiply-Accumulate c=c+a[i]*b[i]

<MLA> Rd, Rf, Rn, Rm @ Rd= RfxRn+Rm

- **MLA R4, R1, R2, R4 @ R4 = R1xR2+R4**

```
                    .TEXT
00001000:E3A05D41   LDR R5,=A
00001004:E59F6028   LDR R6,=B
00001008:E59F7028   LDR R7,=C
0000100C:E3A03003   Mov R3,#3     ;count Of Numbers
00001010:E3A04000   MOV R4,#0
00001014:           Loop:
00001014:E4951004           Ldr R1,[R5],#4
00001018:E4962004           Ldr R2,[R6],#4
0000101C:E0244291           MLA R4,R1,R2,R4     ;multiple R1 And R2 And Store Result In R4

00001020:E4874004           STR R4,[R7],#4
00001024:E2433001           Sub R3,R3,#1
00001028:E3330000           Teq R3,#0         ; Test For Equality
0000102C:1AFFFFF8           Bne Loop
00001030:EF000011           Swi 0x11        ; Interrupt For Termination Of Program

                    .DATA
00001040:           A: .word 1,3,5
0000104C:           B: .word 2,4,6
00001058:           C: .word
```

Microprocessor & Computer Architecture (µpCA)

Multiply & Multiply with Accumulate to Produce 64-bit Result

UMULL	**Unsigned multiply**	**64-bit result**
UMLAL	**Unsigned multiply accumulate**	**64-bit result**
SMULL	**Signed multiply**	**64-bit result**
SMLAL	**Signed multiply accumulate**	**64-bit result**

UMULL R0, R4, R5, R6 ; Multiplies R5 and R6, writes the top 32 bits to R4 ; and the bottom 32 bits to R0

UMLAL R3, R6, R2, R7 ; Multiplies R2 and R7, adds R6, adds R3, writes the ; top 32 bits to R6, and the bottom 32 bits to R3

Microprocessor & Computer Architecture (µpCA)

Multiply & Multiply with Accumulate to Produce 64-bit Result

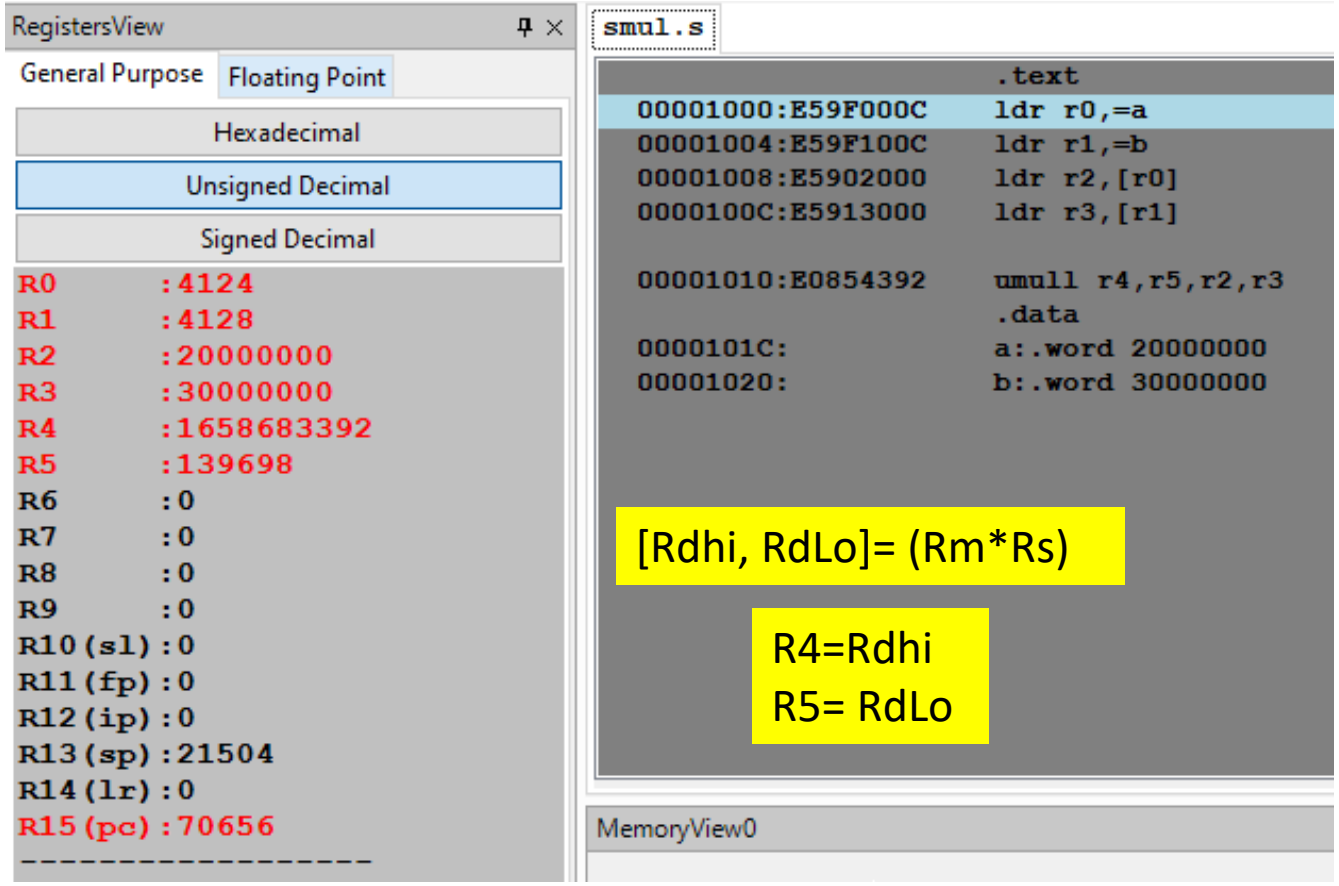
Syntax: <SMLAL/SMULL/UMLAL/UMULL>{cond}{S}**RdLo**, RdHi, Rm,Rs

SMLAL	Signed multiply accumulate Long	[Rdhi, RdLo]=[RdHi,RdLo]+(Rm*Rs)
SMULL	Signed multiply Long	[Rdhi, RdLo]= (Rm*Rs)
UMLAL	Unsigned Multiply accumulate Long	[Rdhi, RdLo]=[RdHi,RdLo]+(Rm*Rs)
UMULL	Unsigned Multiply Long	[Rdhi, RdLo]= (Rm*Rs)

Microprocessor & Computer Architecture (µpCA)

Multiply & Multiply with Accumulate to Produce 64-bit Result

```
.text
ldr r0,=a
ldr r1,=b
ldr r2,[r0]
ldr r3,[r1]

umull r4,r5,r2,r3
.data
a:.word 20000000
b:.word 30000000
```



PSR Instructions

Move to Register from Status Register (MRS): Read from either CPSR or SPSR

Example:

MRS R0, CPSR

MRS R1, SPSR

Move to Status Register from Register (MSR):

Example:

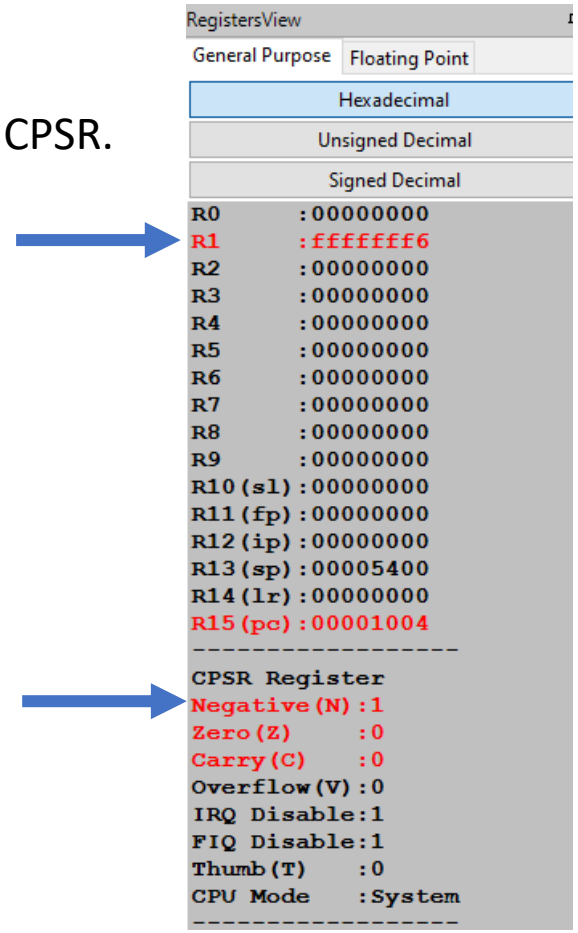
```
MSR CPSR_field, R0
MSR SPSR_field, R1
```

```
_field

_c: Control Field (0:7)
_f: Flag Field(24:31)
_x: Extension (8:15)
_s:Status (16:23)
```

Reference

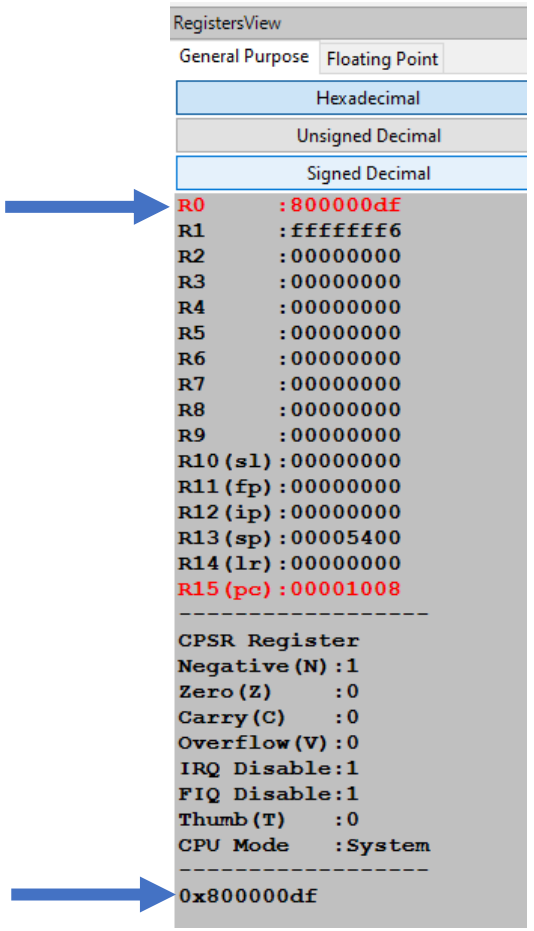
PSR Instructions

```
.text
MOVS R1,#-10
MRS R0,CPSR        ; Take a copy of the CPSR.
AND R0,R0,#0000    ; Clear the mode bits.
MSR CPSR_F,R0      ; Write back the modified CPSR.
.end
```



PSR Instructions

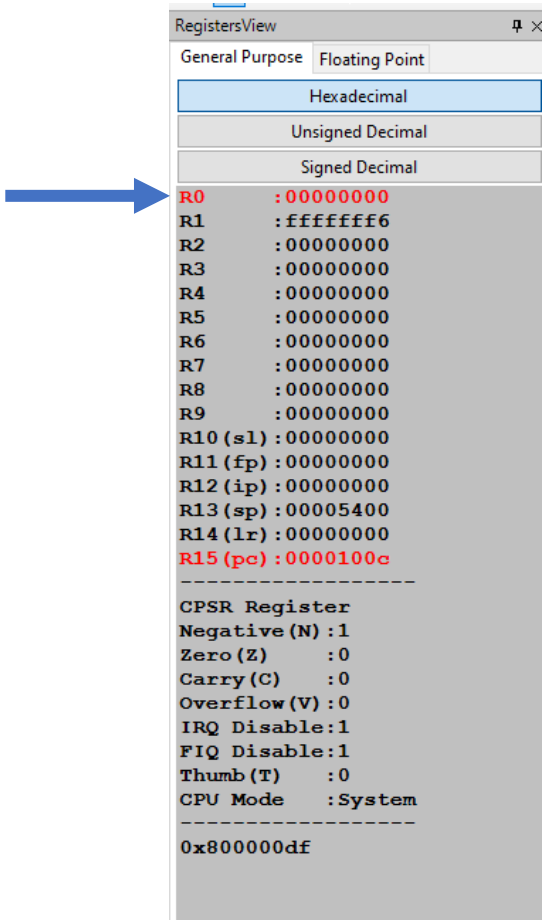
```
.text
MOVS R1,#-10
MRS R0,CPSR        ; Take a copy of the CPSR.
AND R0,R0,#0000    ; Clear the mode bits.
MSR CPSR_F,R0      ; Write back the modified CPSR.
.end
```



Microprocessor & Computer Architecture (µpCA)

PSR Instructions

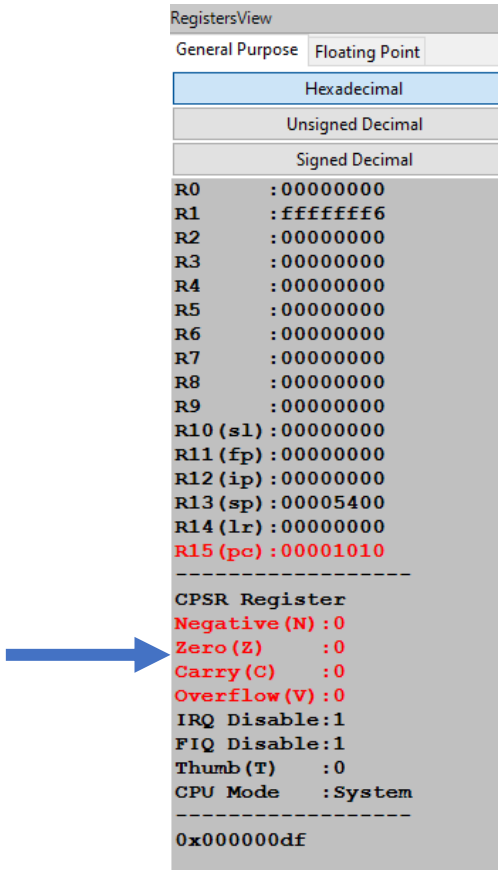
```
.text
MOVS R1,#-10
MRS R0,CPSR        ; Take a copy of the CPSR.
AND R0,R0,#0000    ; Clear the mode bits.
MSR CPSR_F,R0      ; Write back the modified CPSR.
.end
```



Microprocessor & Computer Architecture (µpCA)

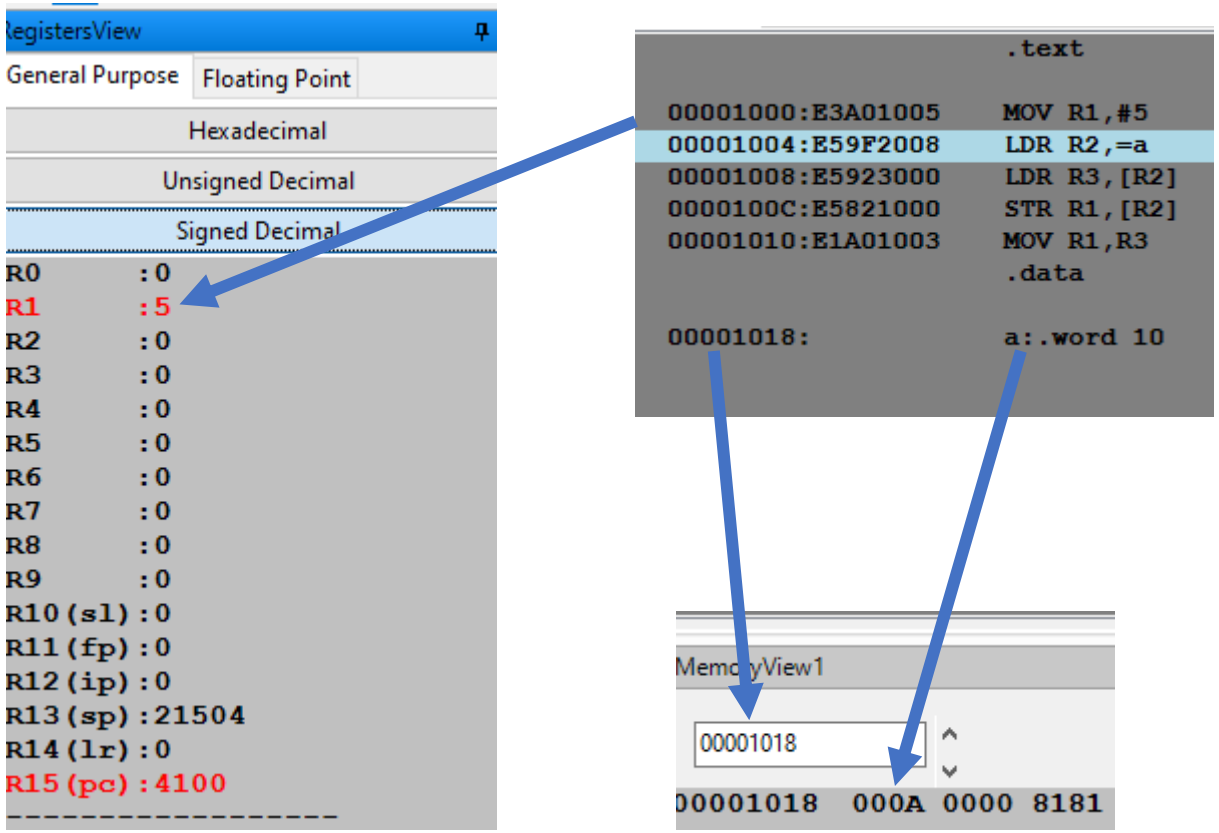
PSR Instructions

```
.text
MOVS R1,#-10
MRS R0,CPSR        ; Take a copy of the CPSR.
AND R0,R0,#0000    ; Clear the mode bits.
MSR CPSR_F,R0      ; Write back the modified CPSR.
.end
```

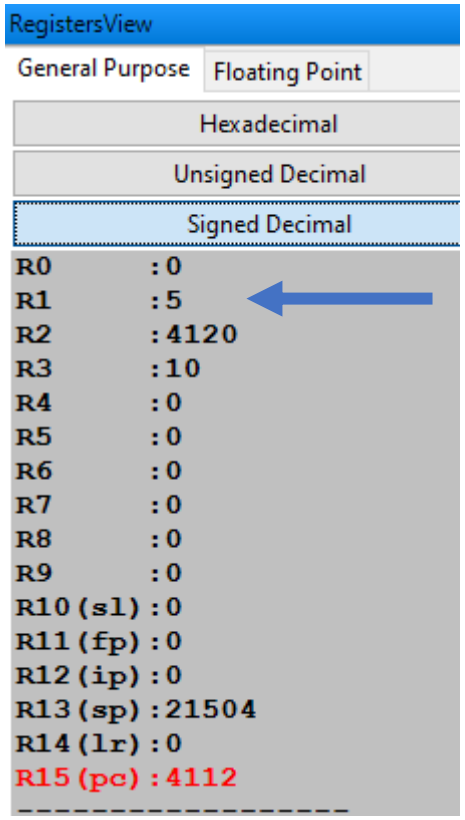


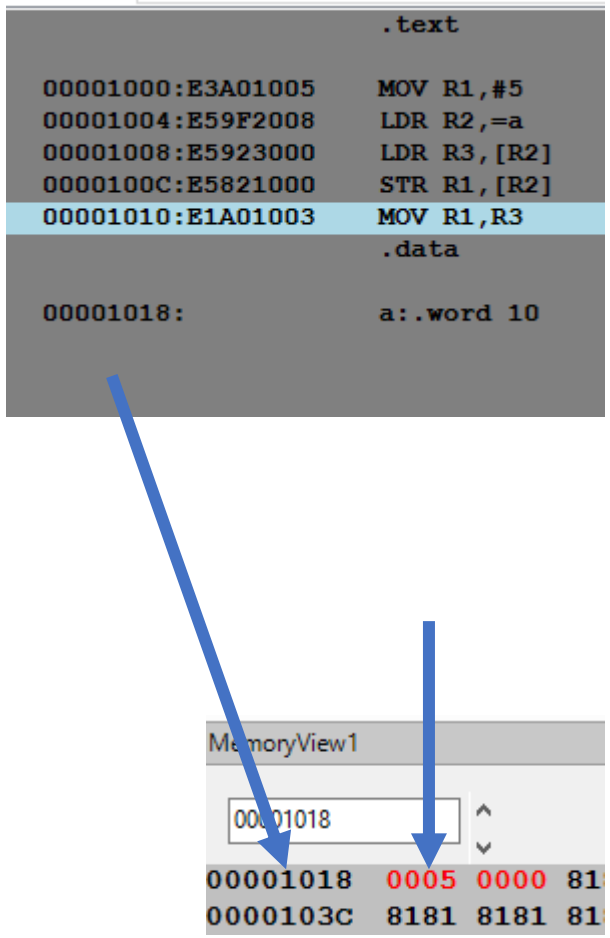
Microprocessor & Computer Architecture (µpCA)

SWAP :A & R1

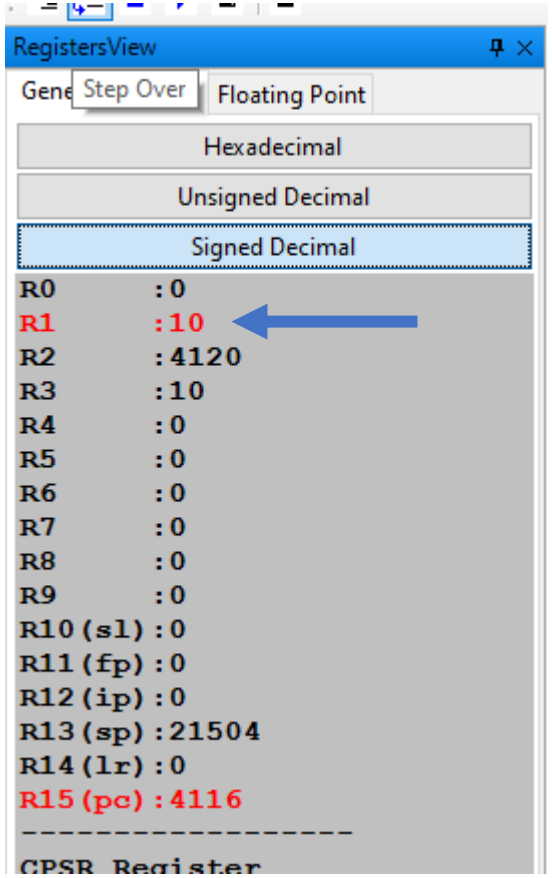


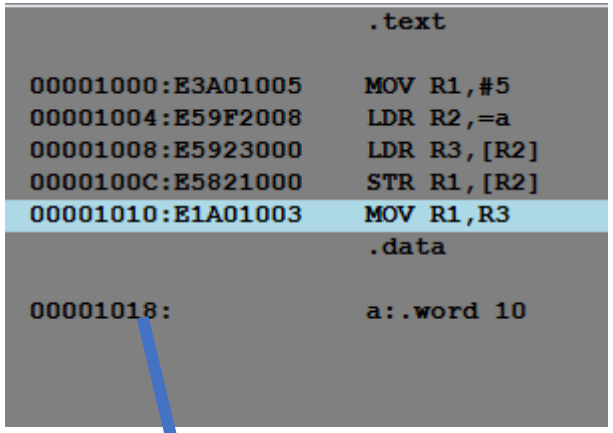
SWAP :A & R1

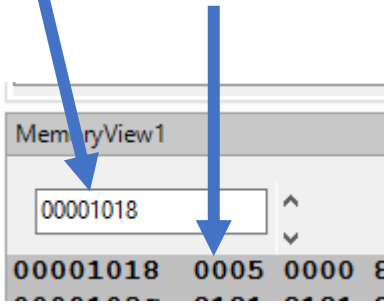

RegistersView
General Purpose
Floating Point
Hexadecimal
Unsigned Decimal
Signed Decimal
R0 :0
R1 :5
R2 :4120
R3 :10
R4 :0
R5 :0
R6 :0
R7 :0
R8 :0
R9 :0
R10(sl):0
R11(fp):0
R12(ip):0
R13(sp):21504
R14(lr):0
R15(pc):4112


.text
00001000:E3A01005 MOV R1,#5
00001004:E59F2008 LDR R2,=a
00001008:E5923000 LDR R3,[R2]
0000100C:E5821000 STR R1,[R2]
00001010:E1A01003 MOV R1,R3
.data
00001018: a:.word 10
MemoryView1
00001018
00001018 0005 0000 81
0000103C 8181 8181 81

SWAP :A & R1


RegistersView
Step Over
Floating Point
Hexadecimal
Unsigned Decimal
Signed Decimal
R0 :0
R1 :10
R2 :4120
R3 :10
R4 :0
R5 :0
R6 :0
R7 :0
R8 :0
R9 :0
R10(sl):0
R11(fp):0
R12(ip):0
R13(sp):21504
R14(lr):0
R15(pc):4116
CPSR Register

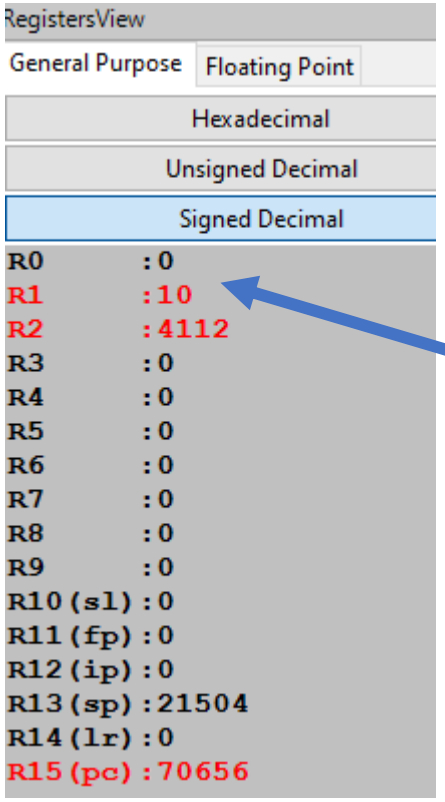

.text
00001000:E3A01005 MOV R1,#5
00001004:E59F2008 LDR R2,=a
00001008:E5923000 LDR R3,[R2]
0000100C:E5821000 STR R1,[R2]
00001010:E1A01003 MOV R1,R3
.data
00001018: a:.word 10

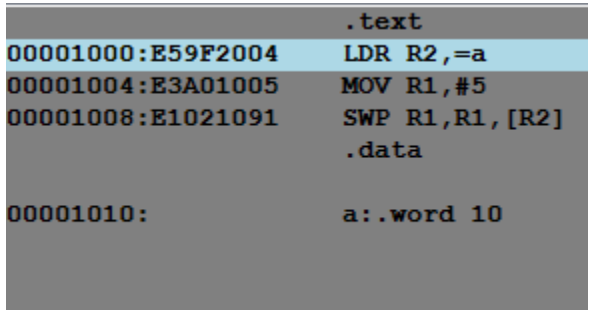

MemoryView1
00001018
00001018 0005 0000

SWAP instruction

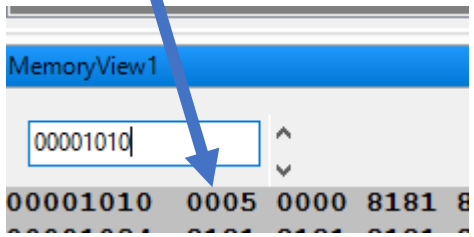
SWP <Swap Destination>, <Original>, [<address>]

```
.text
LDR R2,=a
MOV R1,#5
SWP R1,R1,[R2]
.data
a:.word 10
```

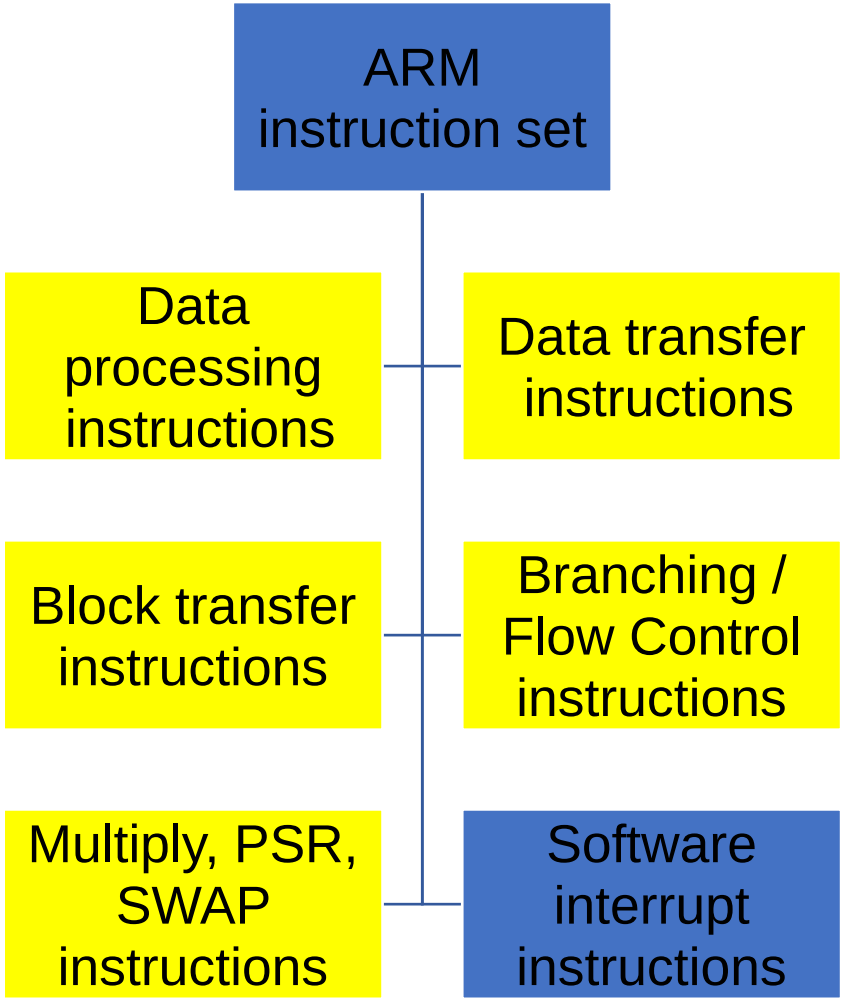




After Swapping



Next Session



THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

INTERRUPTS

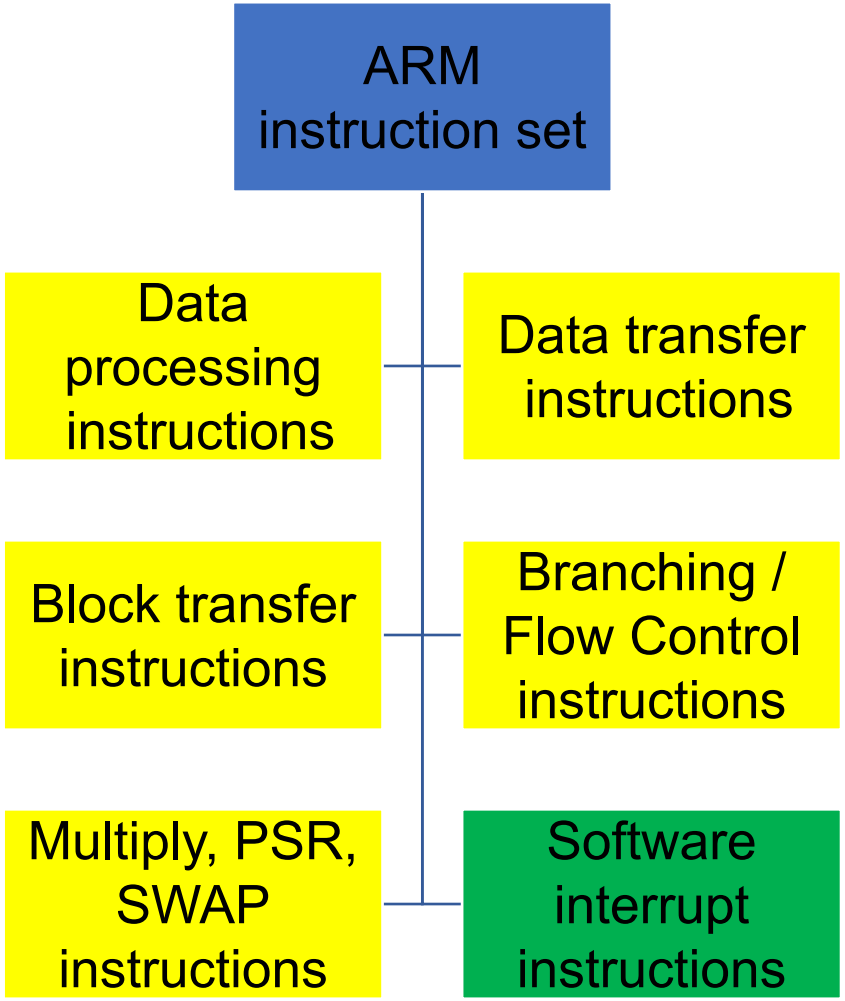
Dr. D. C. Kiran
Department of Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Syllabus

Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ~~ARM Program structure~~
- ~~ARM Instruction Format~~
- **ARM INSTRUCTION SET**
  - ~~Data Processing Instructions~~
  - ~~Flow Control Instructions~~
  - ~~Data Transfer Instructions~~
  - ~~Block Transfer Instructions (Stack & Procedure Call)~~
  - ~~Multiplication~~
  - ~~MSR & MRS Instructions~~
  - ~~Swap~~
  - **Interrupts**


ARM instruction set
Data processing instructions
Data transfer instructions
Block transfer instructions
Branching / Flow Control instructions
Multiply, PSR, SWAP instructions
Software interrupt instructions

What is Interrupt/Exception?

- Main ()
- {
- :
- Doing something
- (e.g.browsing)
- :
- } ring

Can happen anytime
Depends on types of interrupts

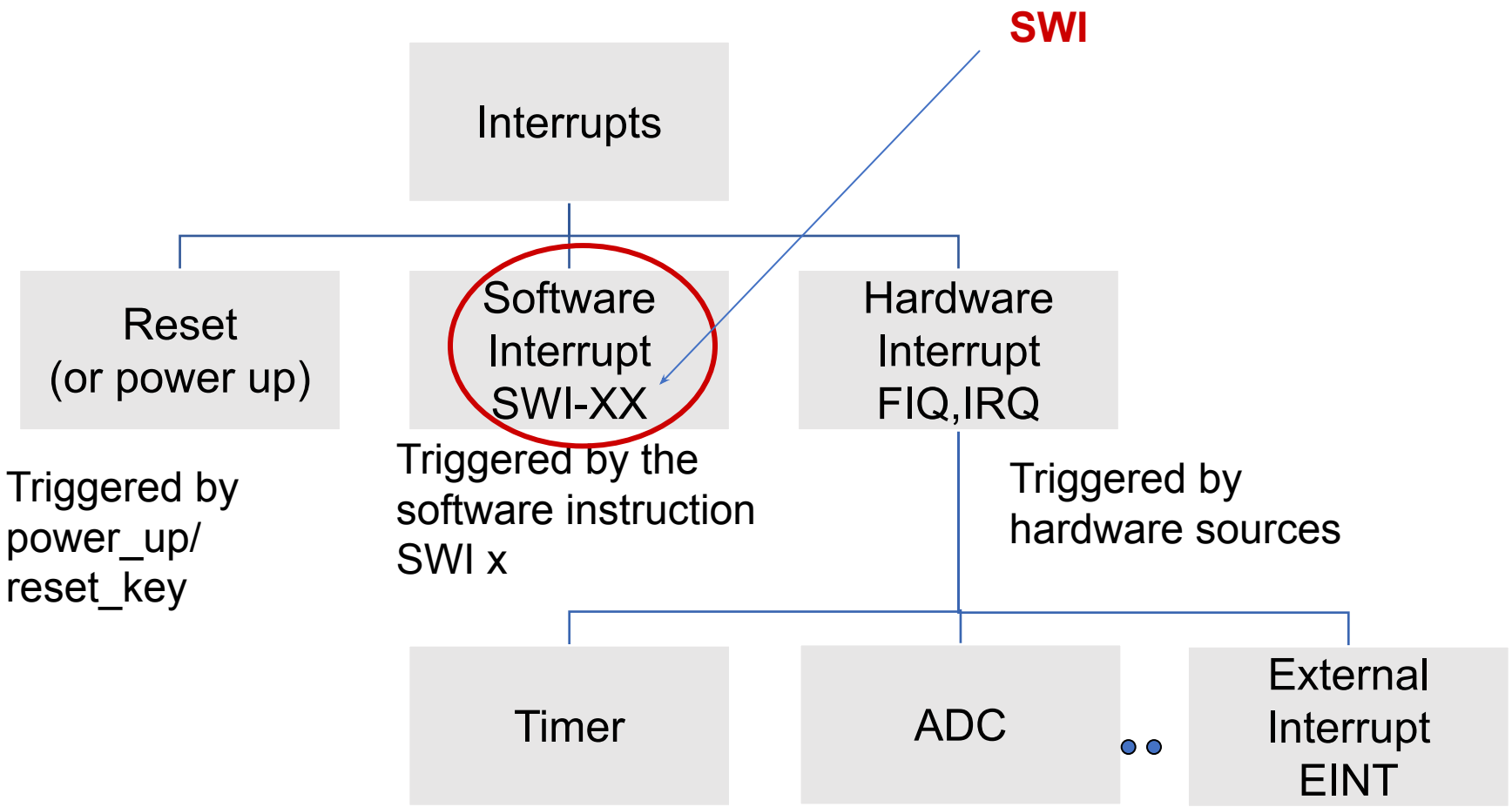
Phone rings

Phone rings

```
_isr() // Interrupt service routine
{
```

```
  some tasks (e.g. answer
                  telephone)

}  //when finished,
   //goes back to main
```

Interrupts


SWI
Interrupts
Reset
(or power up)
Software
Interrupt
SWI-XX
Hardware
Interrupt
FIQ,IRQ
Triggered by
power_up/
reset_key
Triggered by the
software instruction
SWI x
Triggered by
hardware sources
Timer
ADC
External
Interrupt
EINT

- Many sources of “events” during program execution
  - Application makes a system call
  - Software executes instruction illegally (e.g. divides by zero)
  - Peripheral needs attention or has completed a requested action
- How do we know that an event has occurred?
- Broadly, two options to “detect” events
  - Polling
    - We can repeatedly poll the app/processor/peripherals
    - When an event occurs, detect this via a poll and take action
  - Interrupts
    - Let the app/processors/peripheral notify us instead
    - Take action when such a notification occurs (or shortly later)

Reference

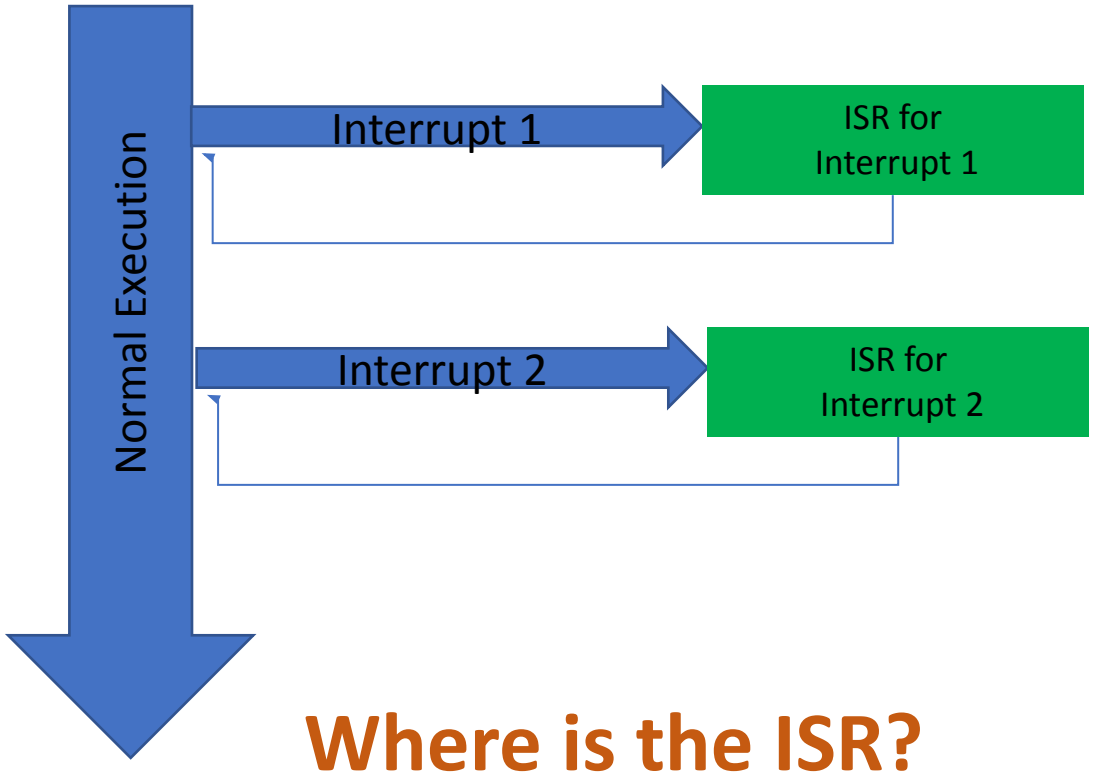
SOFTWARE METHOD – POLLING

- To serve multiple interrupts generated by devices simultaneously
- It's a way to decide to which interrupt will be served first on priority.
- A service program will decide which interrupt to serve based on priority
- All the devices will be checked to see who has generated the interrupt.
- If Flag bit of a device is set, itz interrupt service is served.
- This process is slow

```
if (device[0].flag)
    device[0].service();
else if (device[1].flag)
    device[1].service();
.
.
.
.
.
.
else
    //raise error
```

Event Driven Tasks Execution

Each Event (Interrupt / Exception) has ISR
This is similar to the Sub-Routine call



Where is the ISR?

Somewhere in Code part of Main Memory

Interrupt Service Routine (ISR)

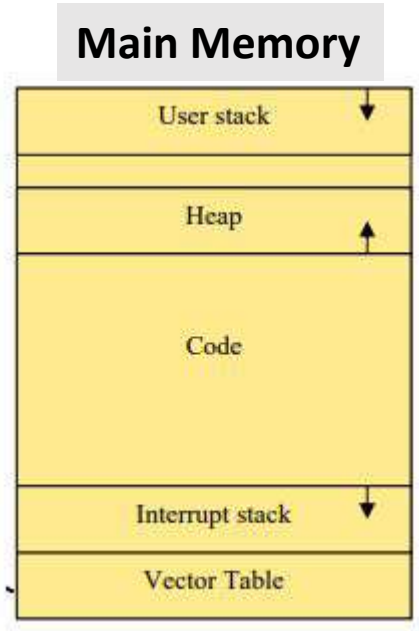
The ISR is a piece of code that's responsible for clearing the source of the interrupt.

ISR is also called device driver in case of the devices and called exception or signal or trap handler in case of software interrupts

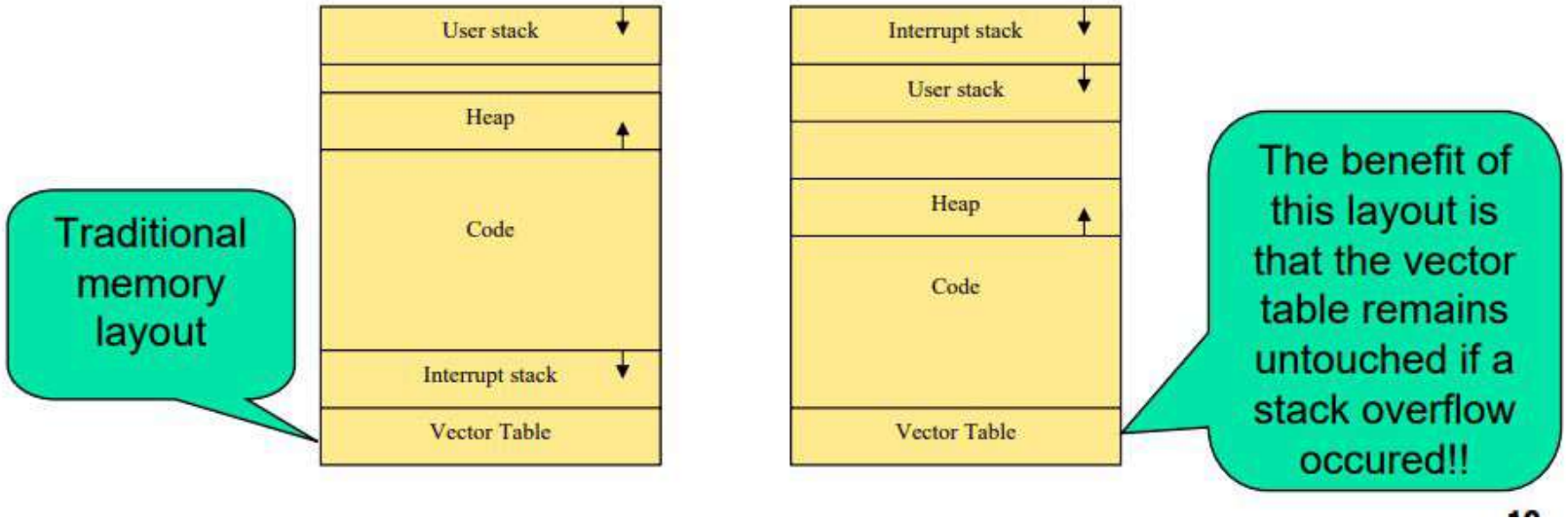
Event Driven Tasks Execution

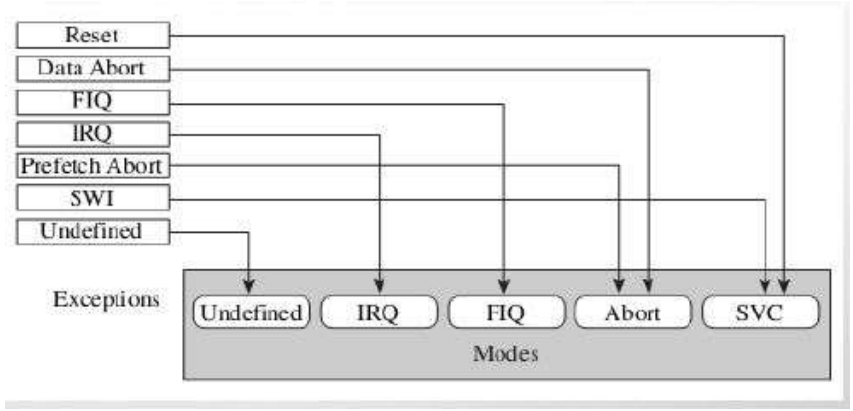
How the processor determines where the ISR is located in code memory for the specific interrupt?

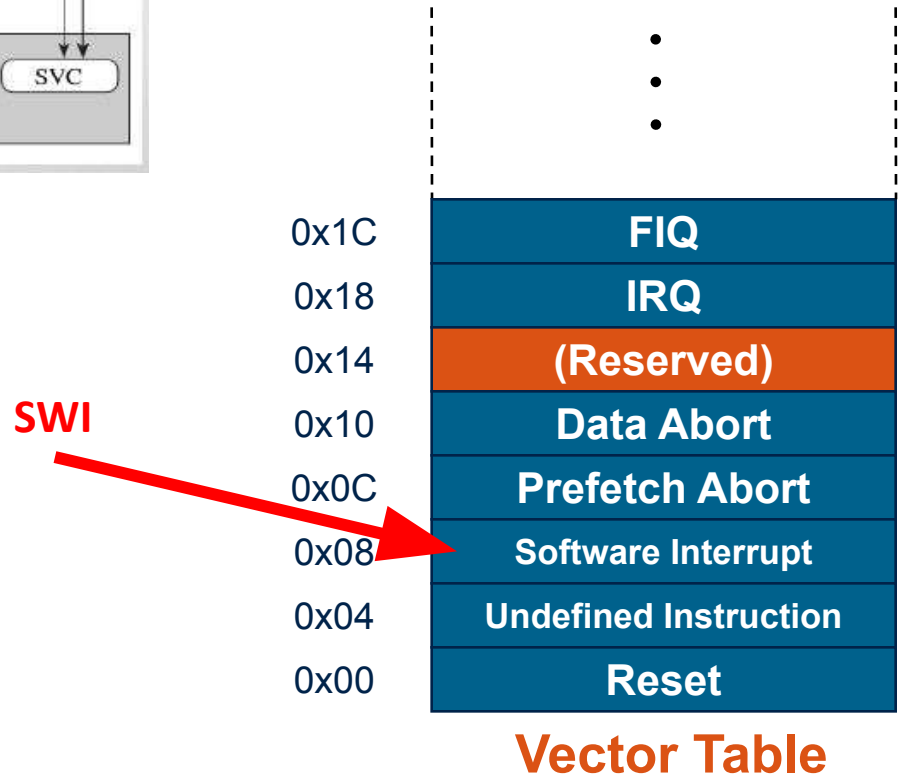
Microprocessor make use of Interrupt Vector Tables to find the starting address of ISR routines.



Reference

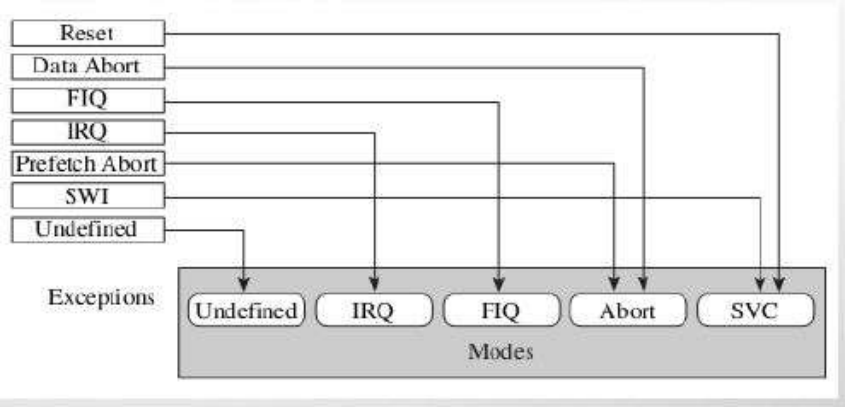

Traditional memory layout
User stack
Heap
Code
Interrupt stack
Vector Table
Interrupt stack
User stack
Heap
Code
Vector Table
The benefit of this layout is that the vector table remains untouched if a stack overflow occured!!





Reference 1

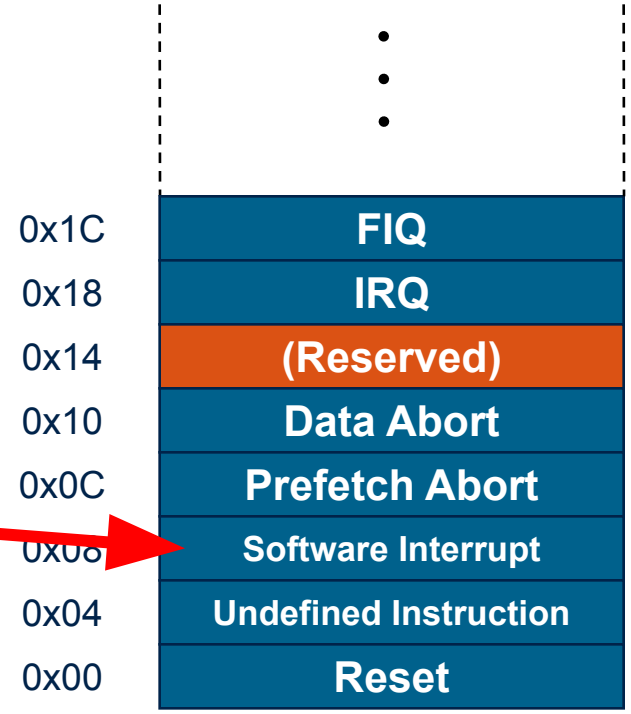
Reference 2



SWI

At this place in Memory, the Branch Instruction to the ISR can be found.

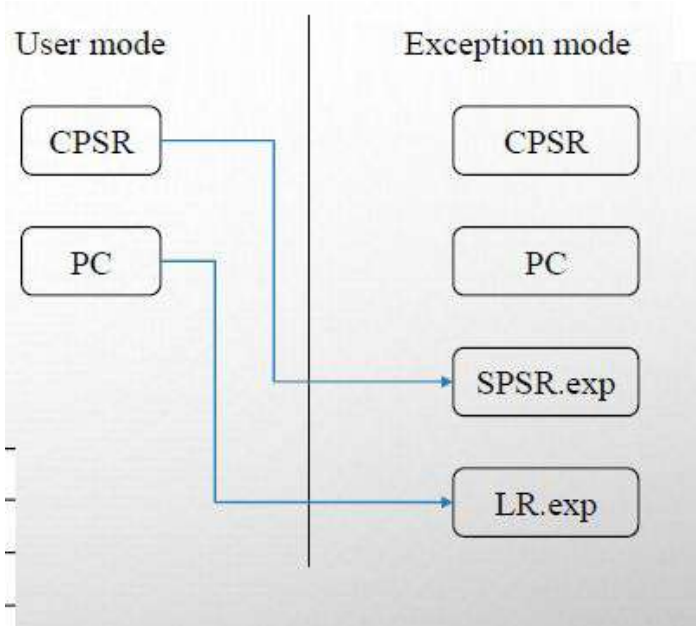
ldr pc, [pc, #_SWI_handler_offset]



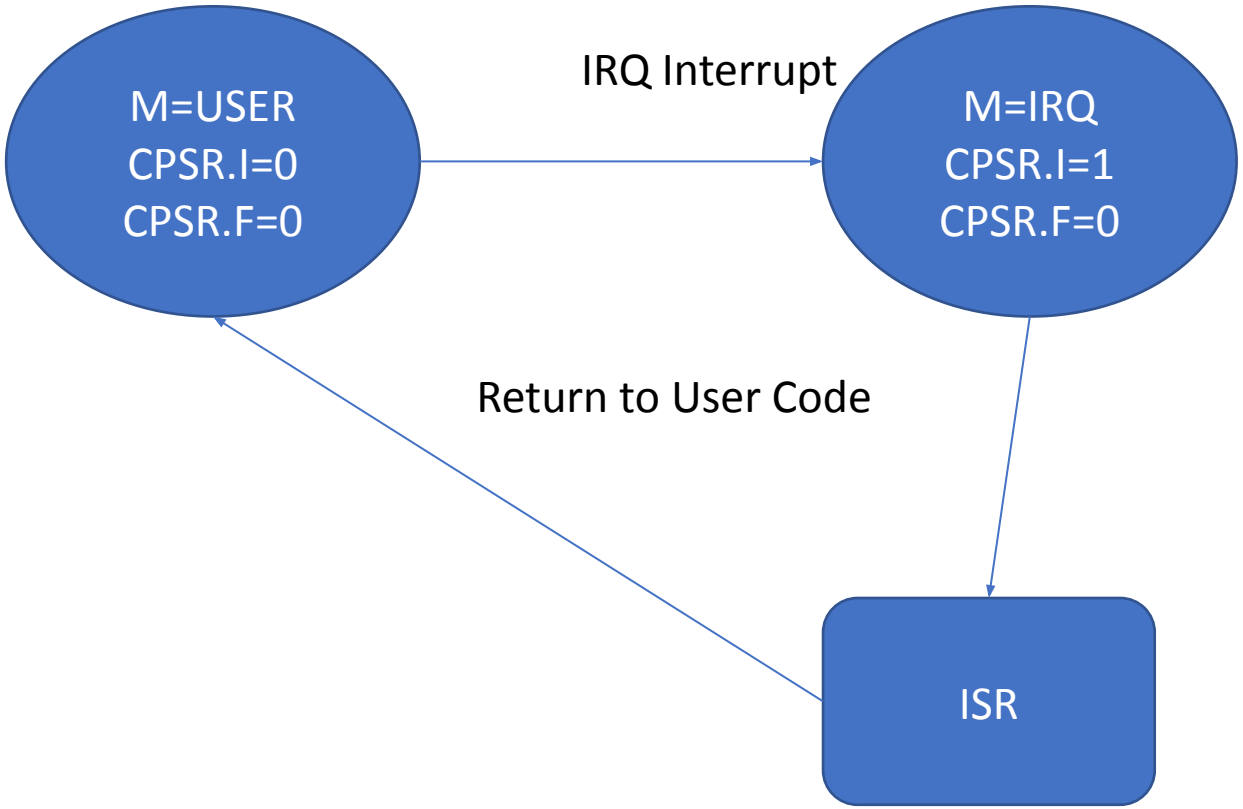
Vector Table

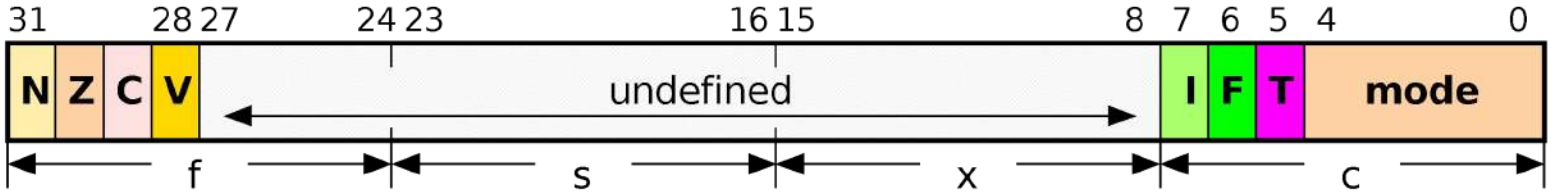
ARM Exception Handling

- Saves the CPSR to the SPSR of the Exception Mode
- Saves the PC to the LR of the Exception mode
- Sets the CPSR to the Exception Mode
- Sets PC to the address of the Exception Handler

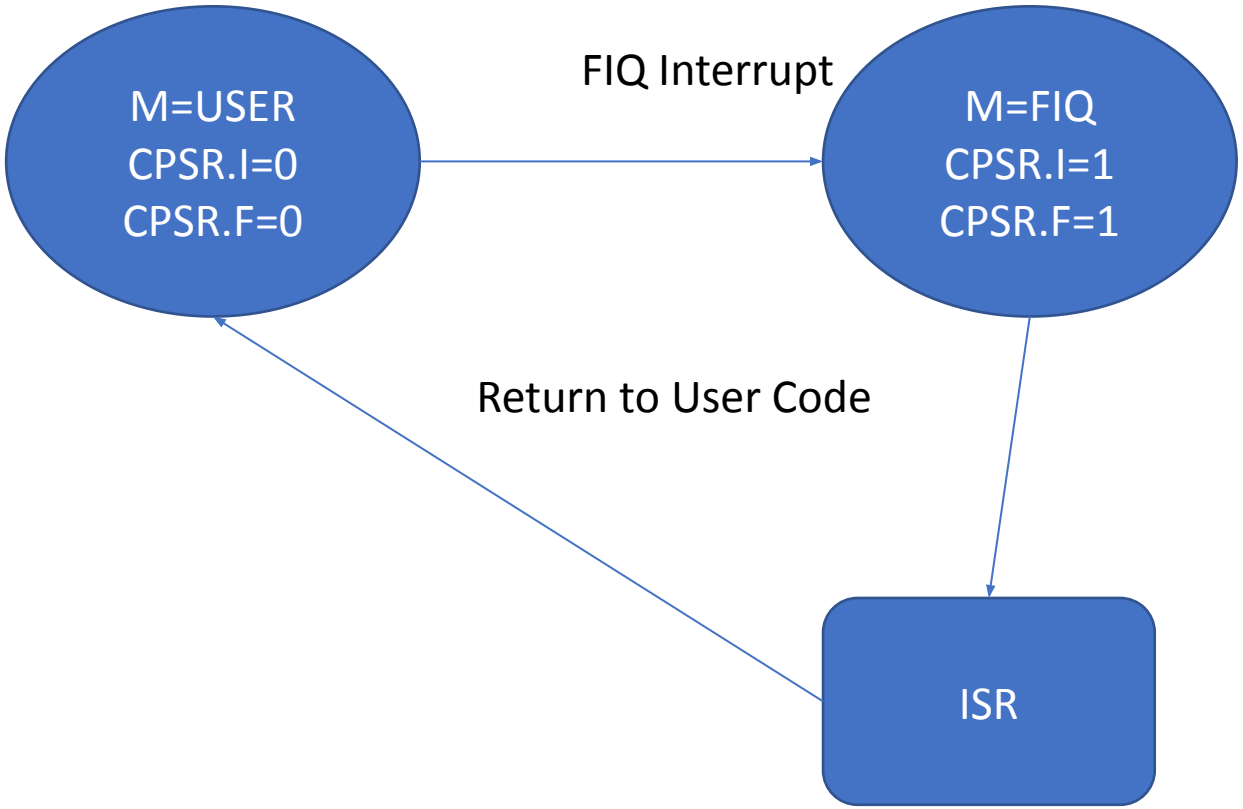


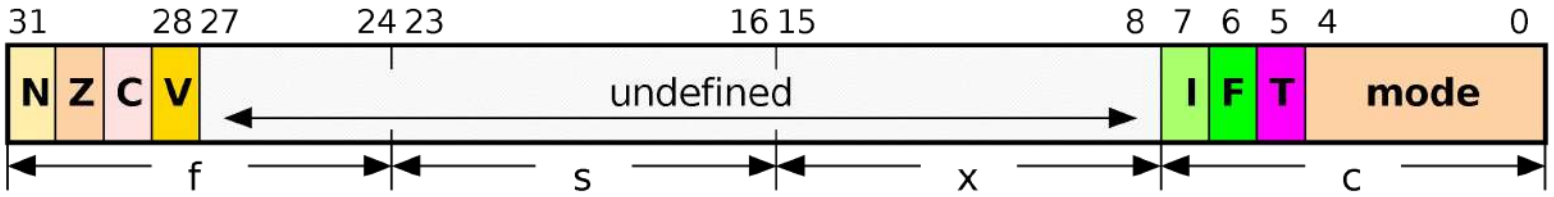
Handling IRQ and FIQ Interrupts





Handling IRQ and FIQ Interrupts





Note the following

This is done by modifying the ***CPSR***, this is done using only 3 ARM instruction:

MRS	To read *CPSR*
MSR	To store in *CPSR*
BIC	Bit clear instruction
ORR	OR instruction

Enabling an IRQ/FIQ Interrupt:

```
MRS   r1, cpsr
BIC   r1, r1, #0x80/0x40
MSR   cpsr_c, r1
```

Disabling an IRQ/FIQ Interrupt:

```
MRS   r1, cpsr
ORR   r1, r1, #0x80/0x40
MSR   cpsr_c, r1
```

I is 7^{th} Bit in CPSR
F is 6^{th} Bit in CPSR

0x80= 128 in binary 2^7
0x40 = 64 in binary 2^6

Enabling and Disabling FIQ and IRQ Exceptions

Enabling an interrupt.

cpsr value	IRQ	FIQ
Pre	*nzcvqjIFt_SVC*	*nzcvqjIFt_SVC*

Code	`enable_irq`
	`MRS r1, cpsr`
	`BIC r1, r1, #0x80`
	`MSR cpsr_c, r1`
		`enable_fiq`
		`MRS r1, cpsr`
		`BIC r1, r1, #0x40`
		`MSR cpsr_c, r1`

Post	*nzcvqjiFt_SVC*	*nzcvqjIft_SVC*

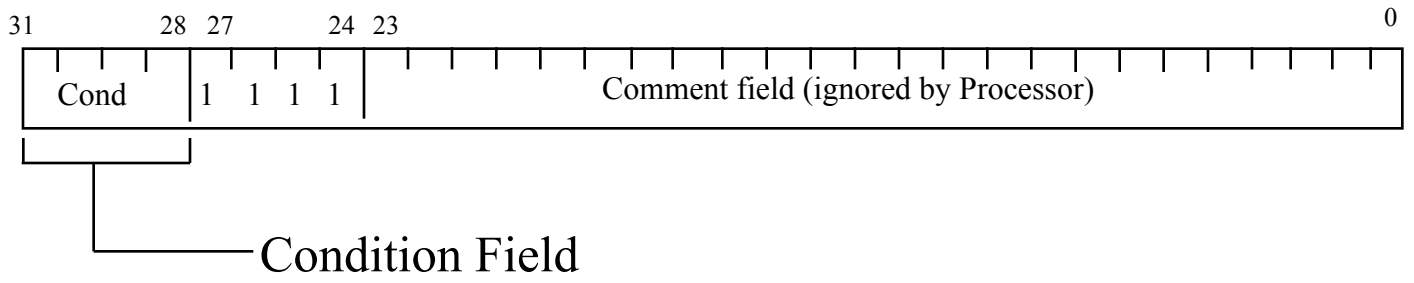
Disabling an interrupt.

cpsr	IRQ	FIQ
Pre	*nzcvqjift_SVC*	*nzcvqjift_SVC*

Code	`disable_irq`
	`MRS r1, cpsr`
	`ORR r1, r1, #0x80`
	`MSR cpsr_c, r1`
		`disable_fiq`
		`MRS r1, cpsr`
		`ORR r1, r1, #0x40`
		`MSR cpsr_c, r1`

Post	*nzcvqjIft_SVC*	*nzcvqjiFt_SVC*

SWI



- In effect, a SWI is a user-defined instruction.
- It causes an exception trap to the SWI hardware vector (thus causing a change to supervisor mode, plus the associated state saving), thus causing the SWI exception handler to be called.
- The handler can then examine the comment field of the instruction to decide what operation has been requested.
- By making use of the SWI mechansim, an operating system can implement a set of privileged operations which applications running in user mode can request.

Microprocessor & Computer Architecture (µpCA)

SWI

Opcode	Description and Action	Inputs	Outputs	EQU
`swi 0x00`	Display Character on Stdout	r0: the character		`SWI_PrChr`
`swi 0x02`	Display String on Stdout	r0: address of a null terminated ASCII string	(see also 0x69 below)	
`swi 0x11`	Halt Execution			`SWI_Exit`
`swi 0x12`	Allocate Block of Memory on Heap	r0: block size in bytes	r0:address of block	`SWI_MeAlloc`
`swi 0x13`	Deallocate All Heap Blocks			`SWI_DAlloc`
`swi 0x66`	Open File (mode values in r1 are: 0 for input, 1 for output, 2 for appending)	r0: file name, i.e. address of a null terminated ASCII string containing the name r1: mode	r0:file handle If the file does not open, a result of -1 is returned	`SWI_Open`
`swi 0x68`	Close File	r0: file handle		`SWI_Close`
`swi 0x69`	Write String to a File or to Stdout	r0: file handleor Stdout r1: address of a null terminated ASCII string		`SWI_PrStr`

SWI

Opcode	Description and Action	Inputs	Outputs	EQU
`swi 0x6a`	Read String from a File	r0: file handle r1: destination address r2: max bytes to store	r0: number of bytes stored	`SWI_RdStr`
`swi 0x6b`	Write Integer to a File	r0: file handle r1: integer		`SWI_PrInt`
`swi 0x6c`	Read Integer from a File	r0: file handle	r0: the integer	`SWI_RdInt`
`swi 0x6d`	Get the current time (ticks)		r0: the number of ticks (milliseconds)	`SWI_Timer`

A procedure to compute the statement in high level language using ARM ALP.

```
if (R0==R1) R2++;
```

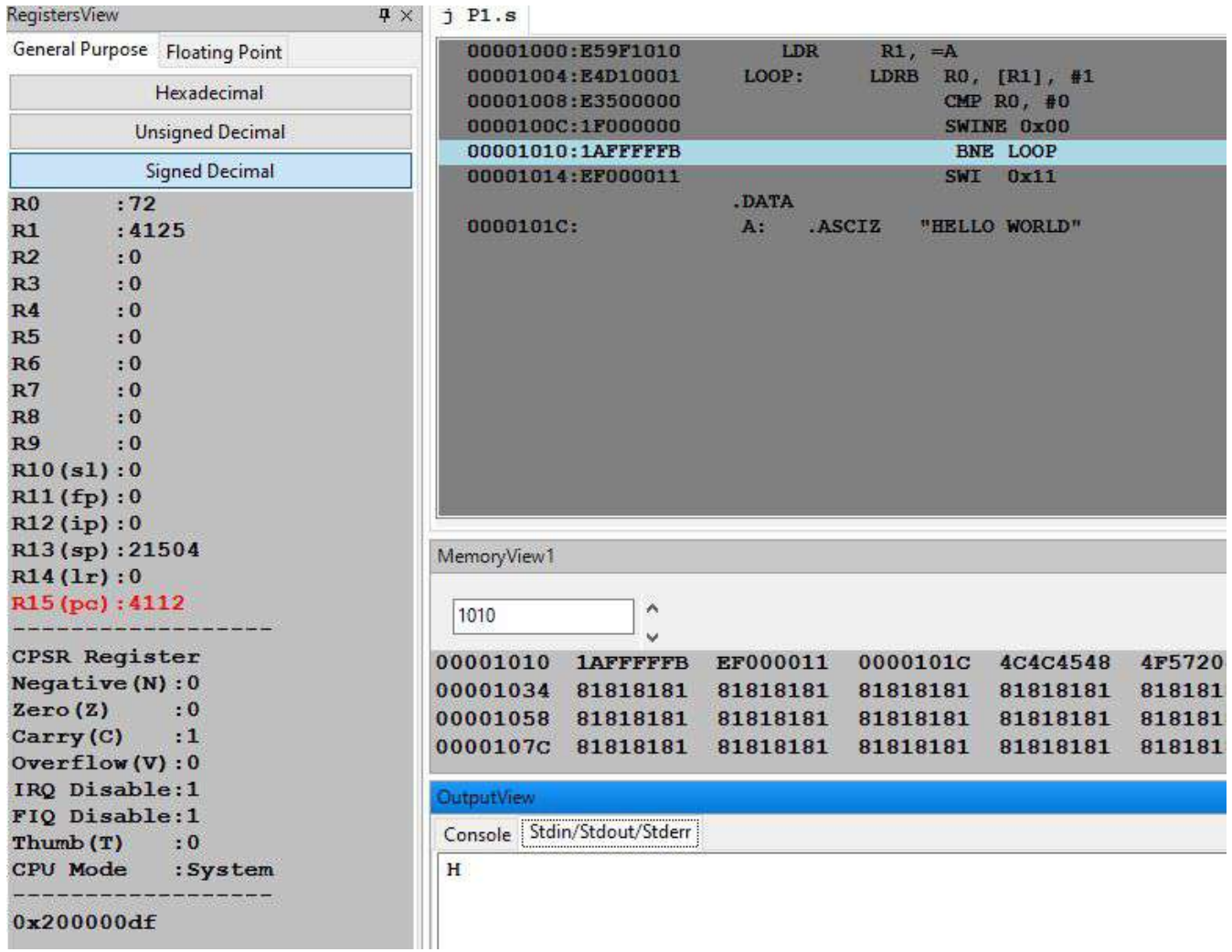
```
MOV R0, #10
MOV R1, #10
BL GREAT
SWI 0x11        ; terminate the program / logical end.

GREAT:  CMP   R0, R1
        ADDEQ R2, R2, #1
        MOV PC, LR
```

A program to display a string on the screen using ARM ALP.

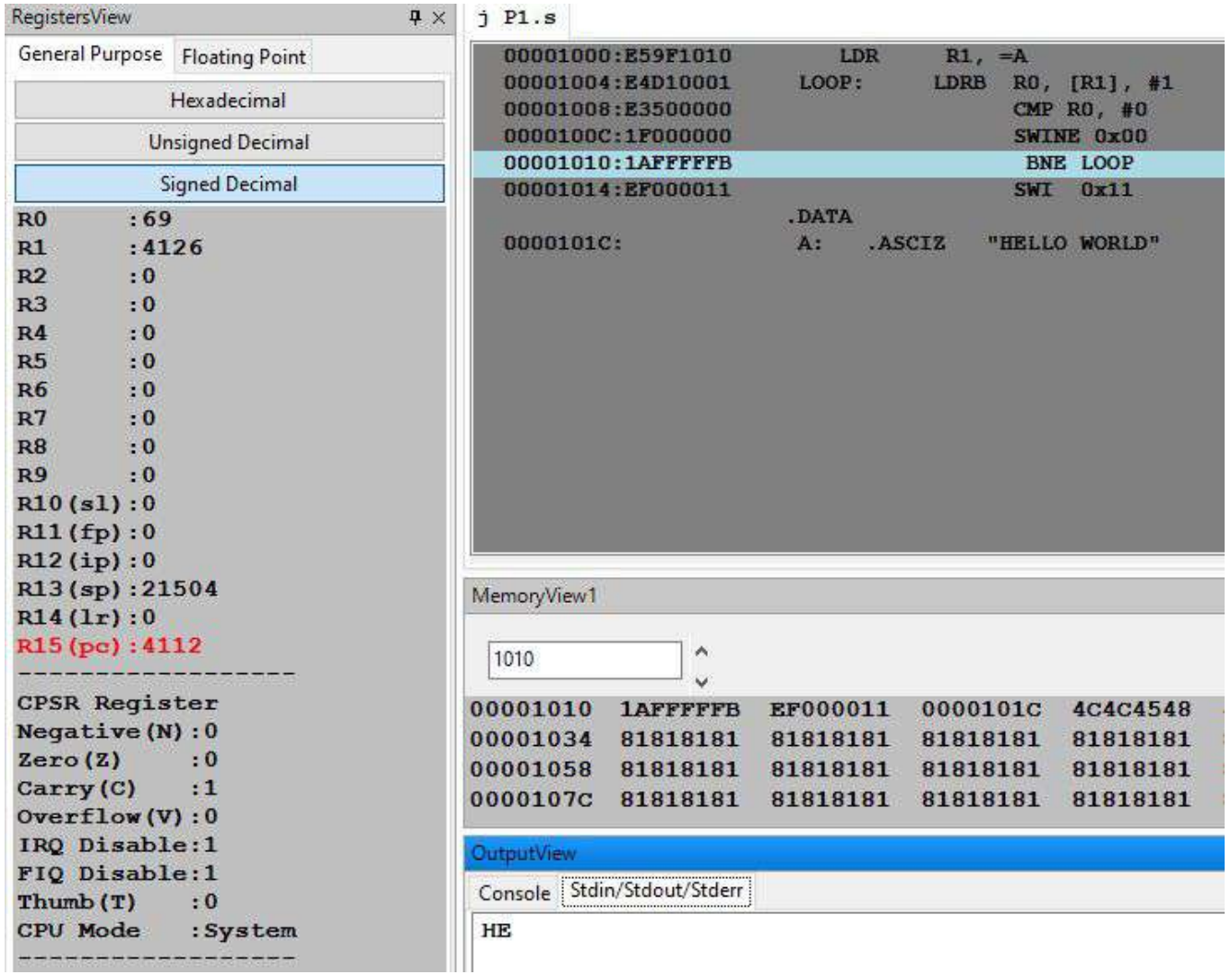
printf(" Hello World");

```
        LDR    R1, =A
LOOP:   LDRB  R0, [R1], #1
        CMP R0, #0
        SWINE 0x00          ; display a character on the screen.
          BNE LOOP
        SWI  0x11           ; terminate the program.
.DATA
A:  .ASCIZ  "HELLO WORLD"
```

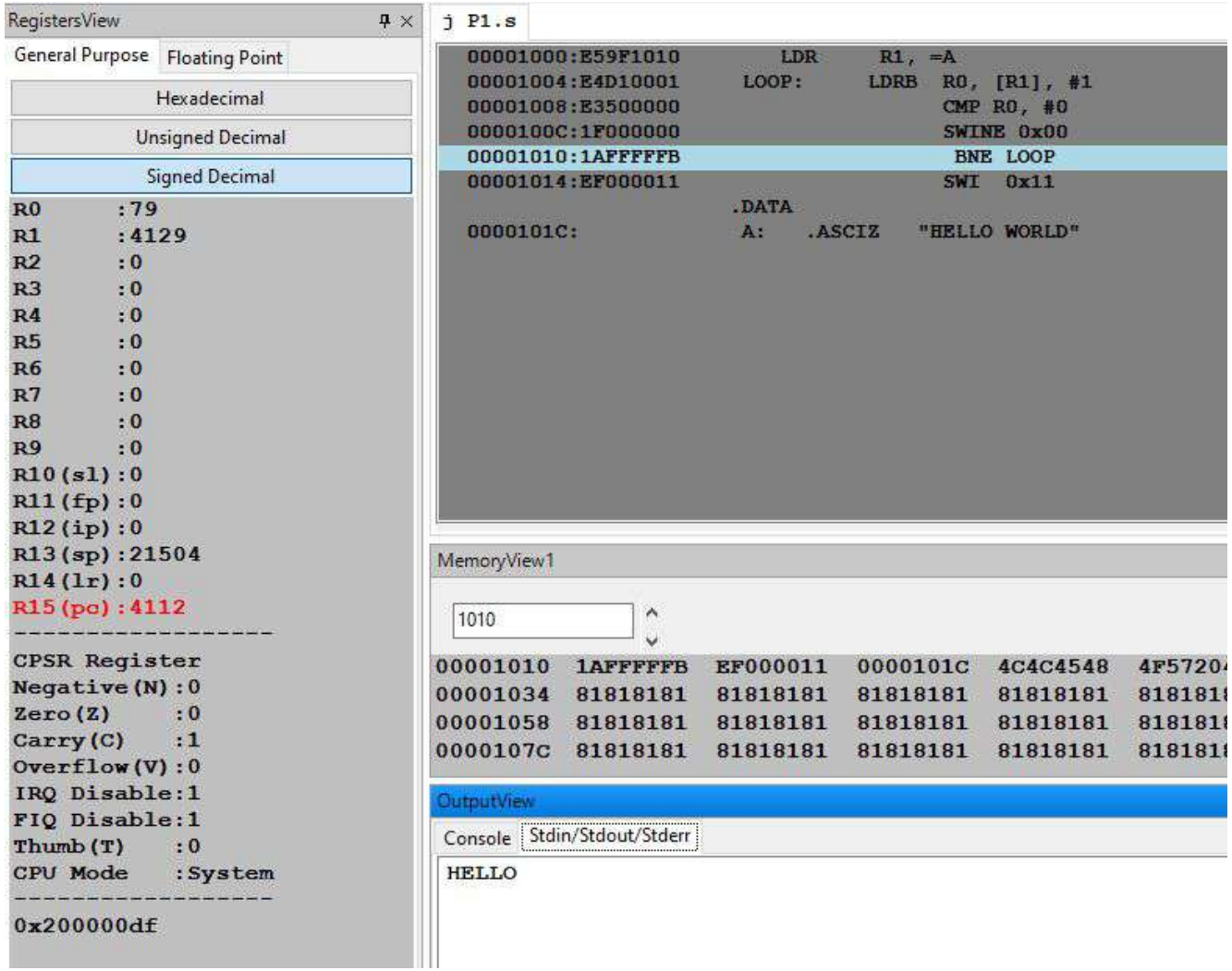
Microprocessor & Computer Architecture (µpCA)



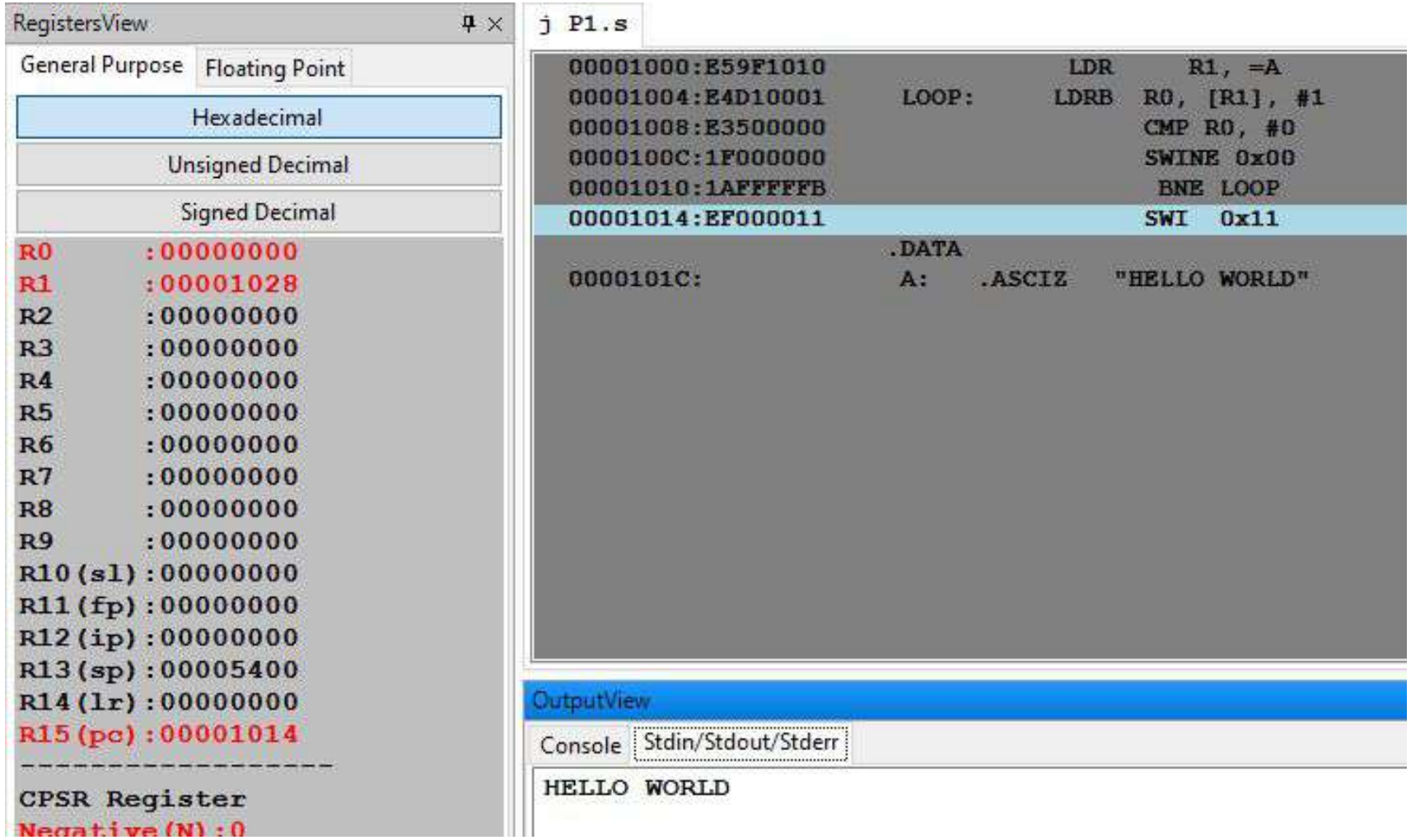
Microprocessor & Computer Architecture (µpCA)



Microprocessor & Computer Architecture (µpCA)



Microprocessor & Computer Architecture (µpCA)



A procedure to display a string on the screen using ARM ALP.

// printf (" Hello World");

```
    LDR    R0, =A
    SWI    0x02        ; display a string on the screen
    SWI    0x11
.DATA

A:  .ASCIZ  "HELLO WORLD"
```

```
.equ SWI_Open, 0x66 ;open a file

.equ SWI_Close,0x68 ;close a file

.equ SWI_PrChr,0x00 ; Write 1 byte to file handle

.equ SWI_RdBytes, 0x6a ; Read n bytes from file handle

.equ SWI_WrBytes, 0x69 ; Write n bytes to file handle

.equ Stdin, 0 ; 0 is the file descriptor for STDIN

.equ Stdout, 1 ; Set output target to be Stdout

.equ SWI_Exit, 0x11 ; Stop execution
```

Microprocessor & Computer Architecture (µpCA)

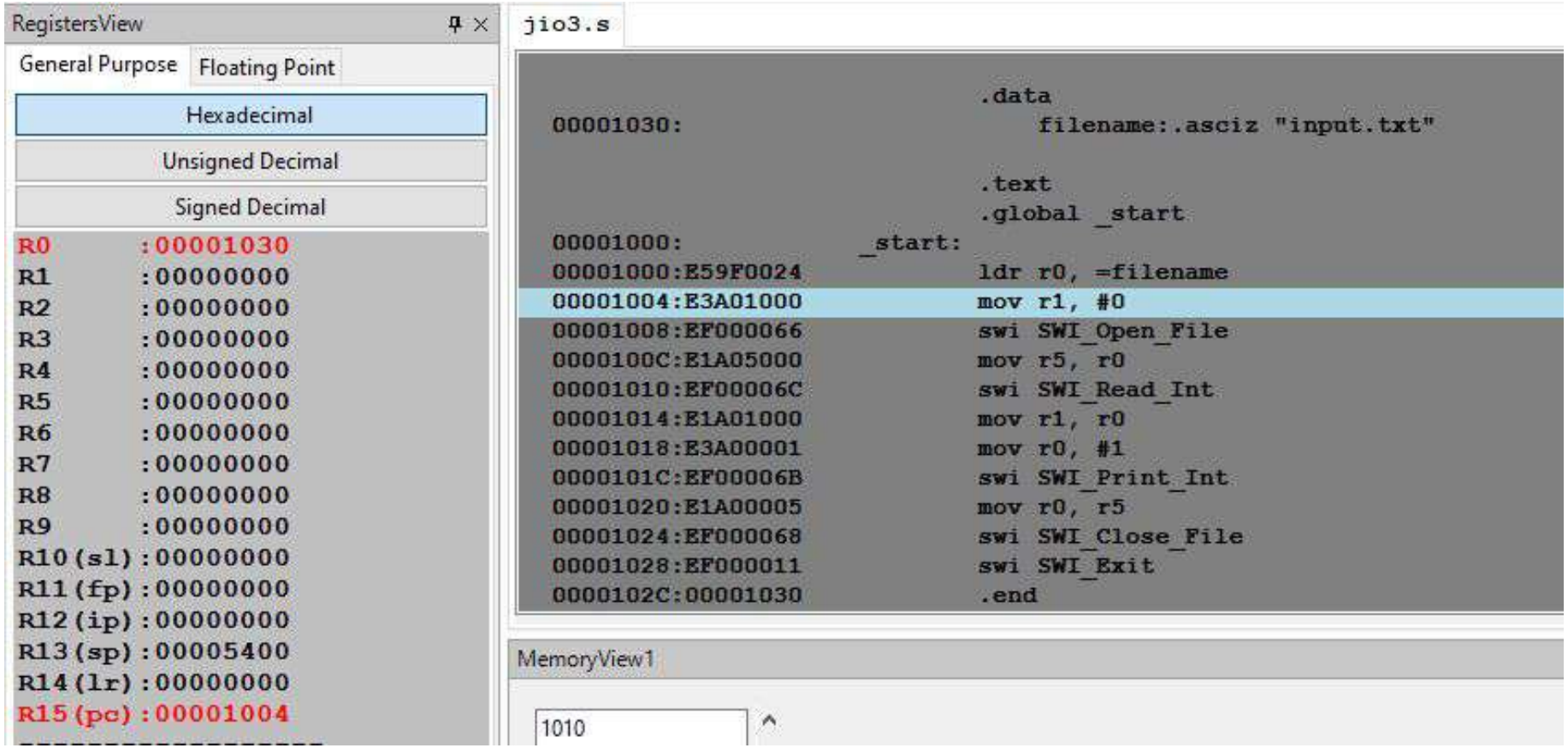
Program to Read an Integer from file and Display

```
.equ SWI_Open_File, 0x66
.equ SWI_Read_Int, 0x6C
.equ SWI_Print_Int, 0x6B
.equ SWI_Close_File, 0x68
.equ SWI_Exit, 0x11
```

```
        .data
          filename:.asciz "input.txt"

        .text
        .global _start
_start:
        ldr r0, =filename
        mov r1, #0
        swi SWI_Open_File
        mov r5, r0
        swi SWI_Read_Int
        mov r1, r0
        mov r0, #1
        swi SWI_Print_Int
        mov r0, r5
        swi SWI_Close_File
        swi SWI_Exit
        .end
```

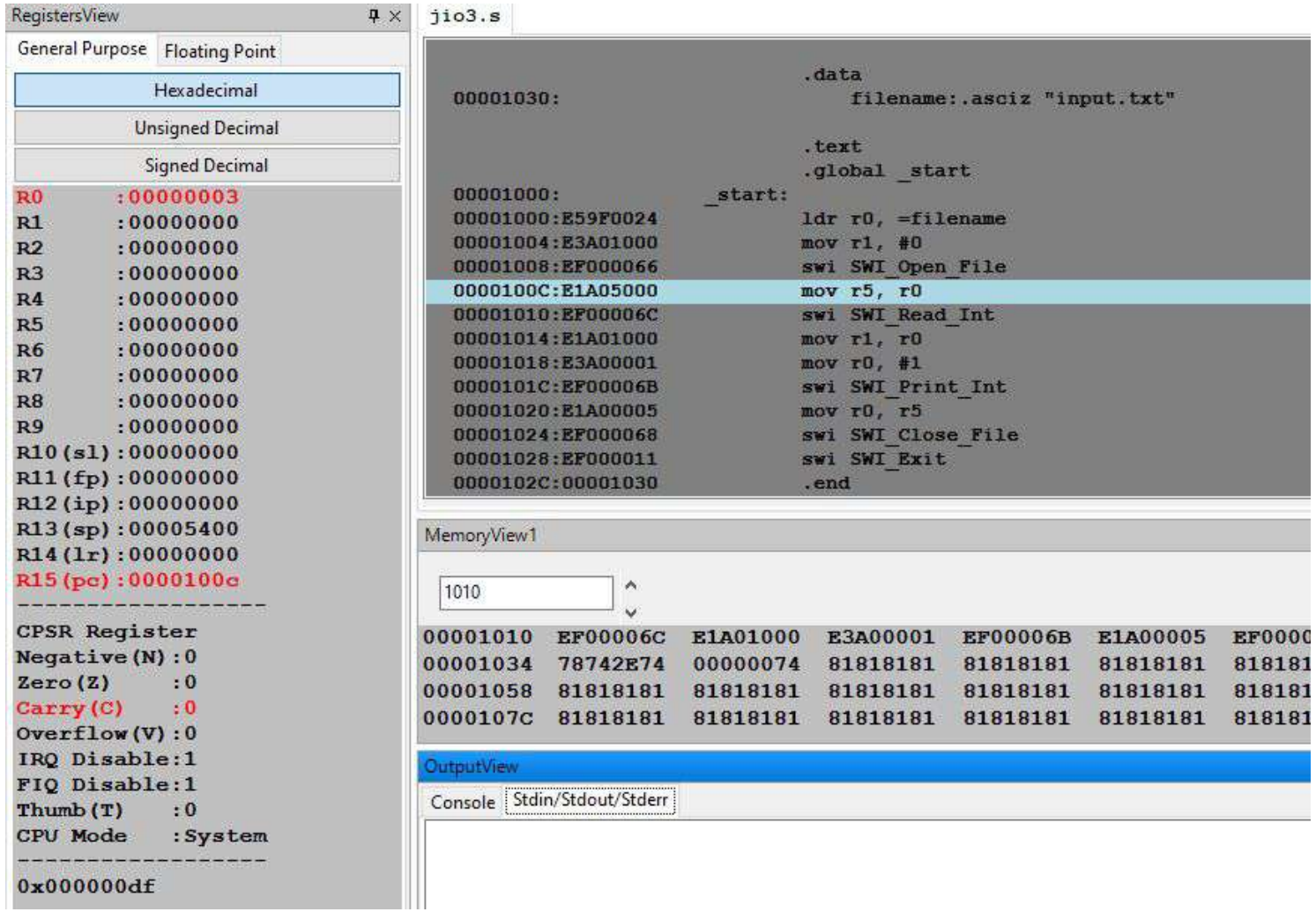
Microprocessor & Computer Architecture (µpCA)

Program to Read an Integer from file and Display



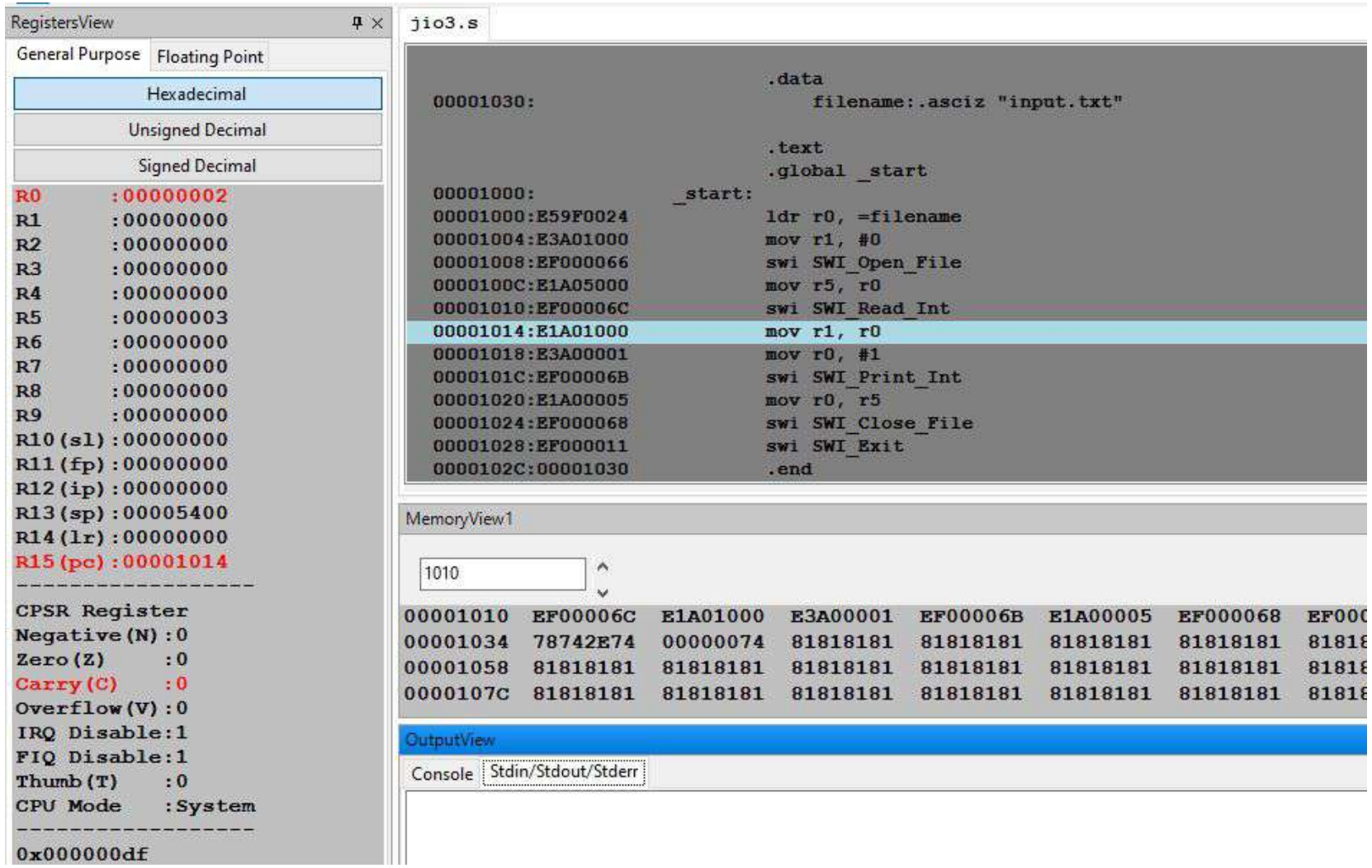
Microprocessor & Computer Architecture (µpCA)

Program to Read an Integer from file and Display



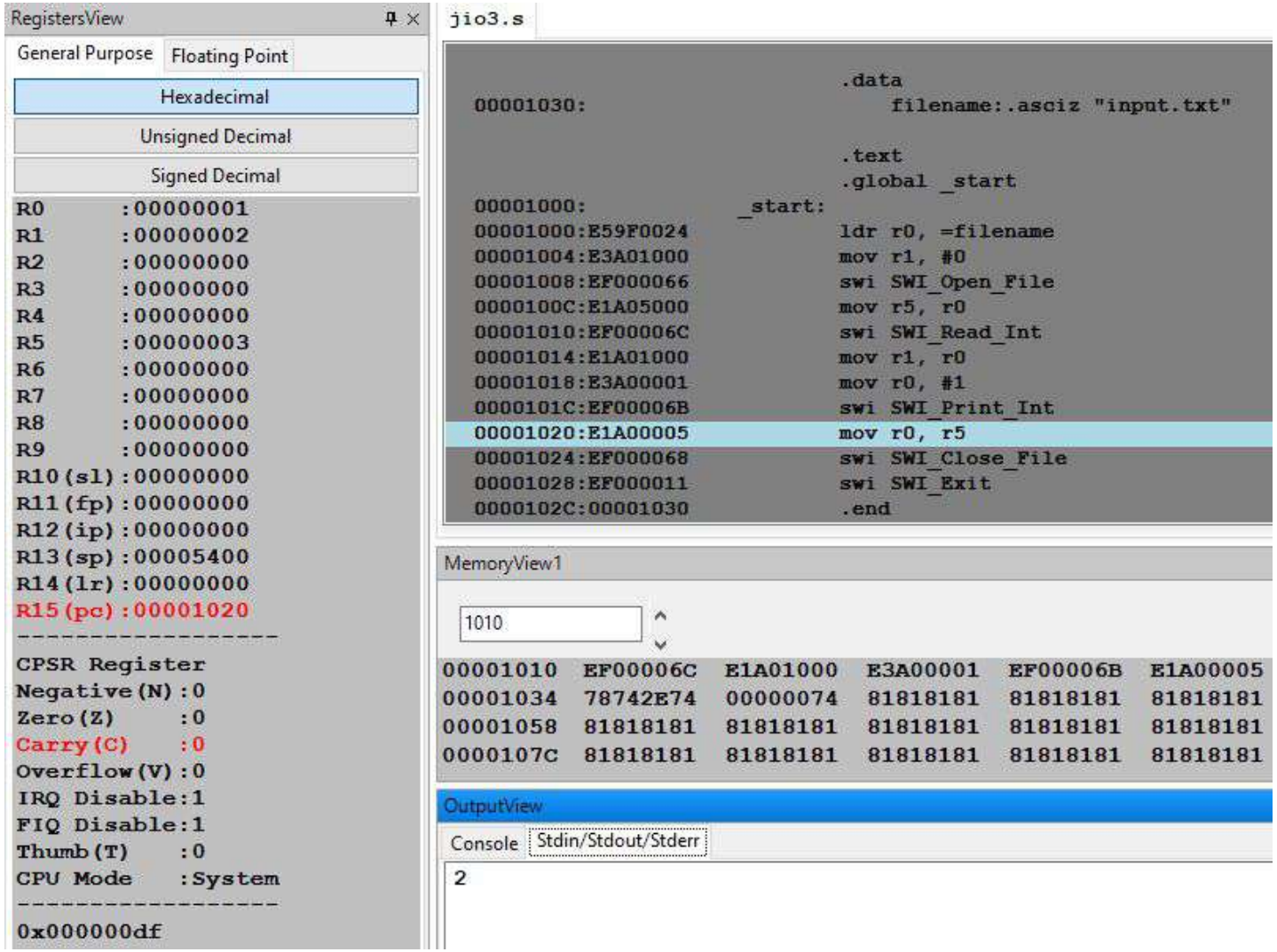
Microprocessor & Computer Architecture (µpCA)

Program to Read an Integer from file and Display



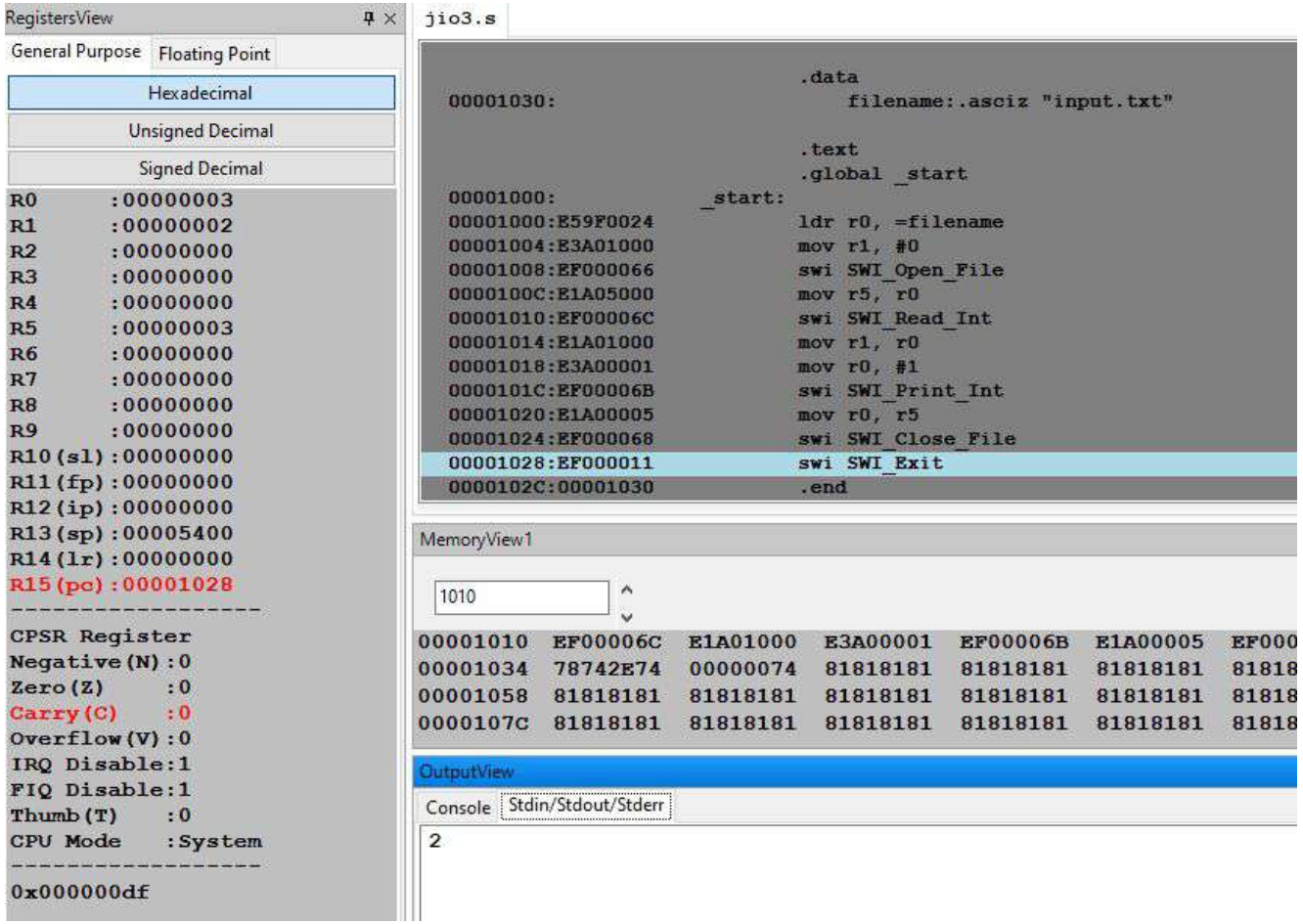
Microprocessor & Computer Architecture (µpCA)

Program to Read an Integer from file and Display



Microprocessor & Computer Architecture (µpCA)

Program to Read an Integer from file and Display



Next Session

Instruction Encoding

THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Instruction Encoding

Dr. D. C. Kiran
Department of Computer Science and Engineering

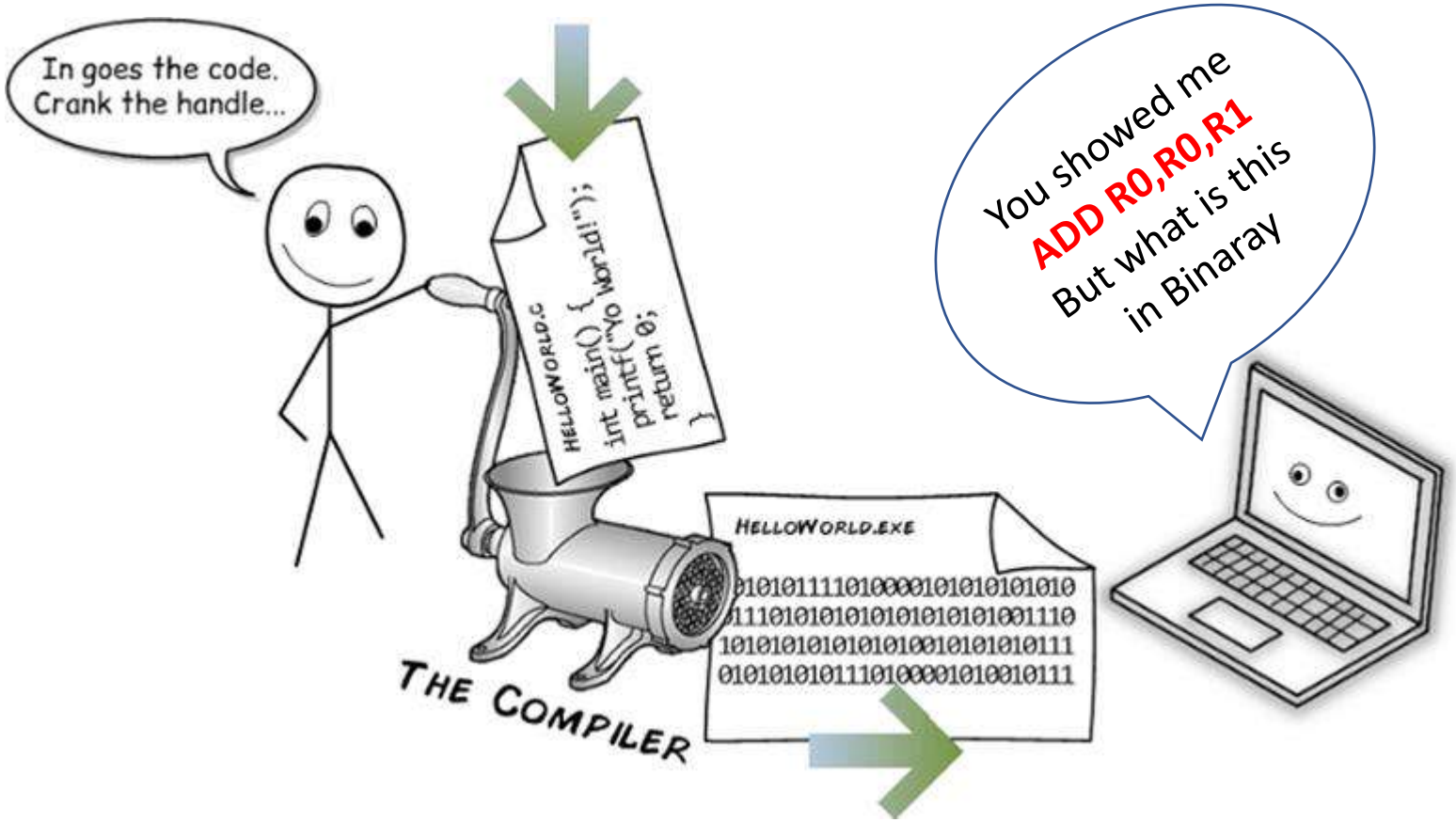
Microprocessor & Computer Architecture (µpCA)

Syllabus

Unit 1: Basic Processor Architecture and Design

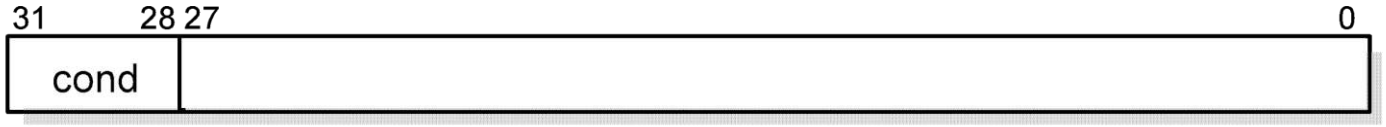
- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ~~ARM Program structure~~
- ~~ARM Instruction Format~~
- ~~ARM INSTRUCTION SET~~
- **Instruction Encoding**
  - ✓ **Instruction Layout**
  - ✓ **Data Processing Instruction**

Microprocessor & Computer Architecture (µpCA)

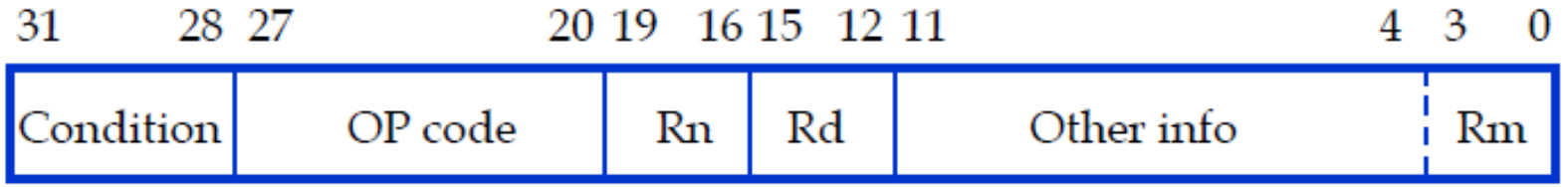


What is an ARM Instruction?

```
OPcode{condition}{S} Rd,Operand1,Operand2
```



Example: Data Processing Instruction

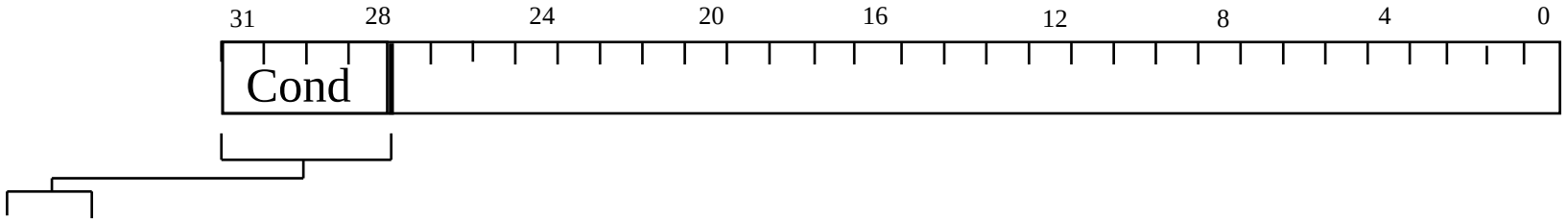


An instruction specifies a **Conditional Execution Code** (Condition), the **OP Code**, **Two or Three Registers** (Rn, Rd, and Rm), and some other information

Reference

Microprocessor & Computer Architecture (µpCA)

The Condition Field



0000 = EQ - Z set (equal)

0001 = NE - Z clear (not equal)

0010 = HS / CS - C set (unsigned higher or same)

0011 = LO / CC - C clear (unsigned lower)

0100 = MI -N set (negative)

0101 = PL - N clear (positive or zero)

0110 = VS - V set (overflow)

0111 = VC - V clear (no overflow)

1000 = HI - C set and Z clear (unsigned higher)

1001 = LS - C clear or Z (set unsigned lower or same)

1010 = GE - N set and V set, or N clear and V clear (>or =)

1011 = LT - N set and V clear, or N clear and V set (>)

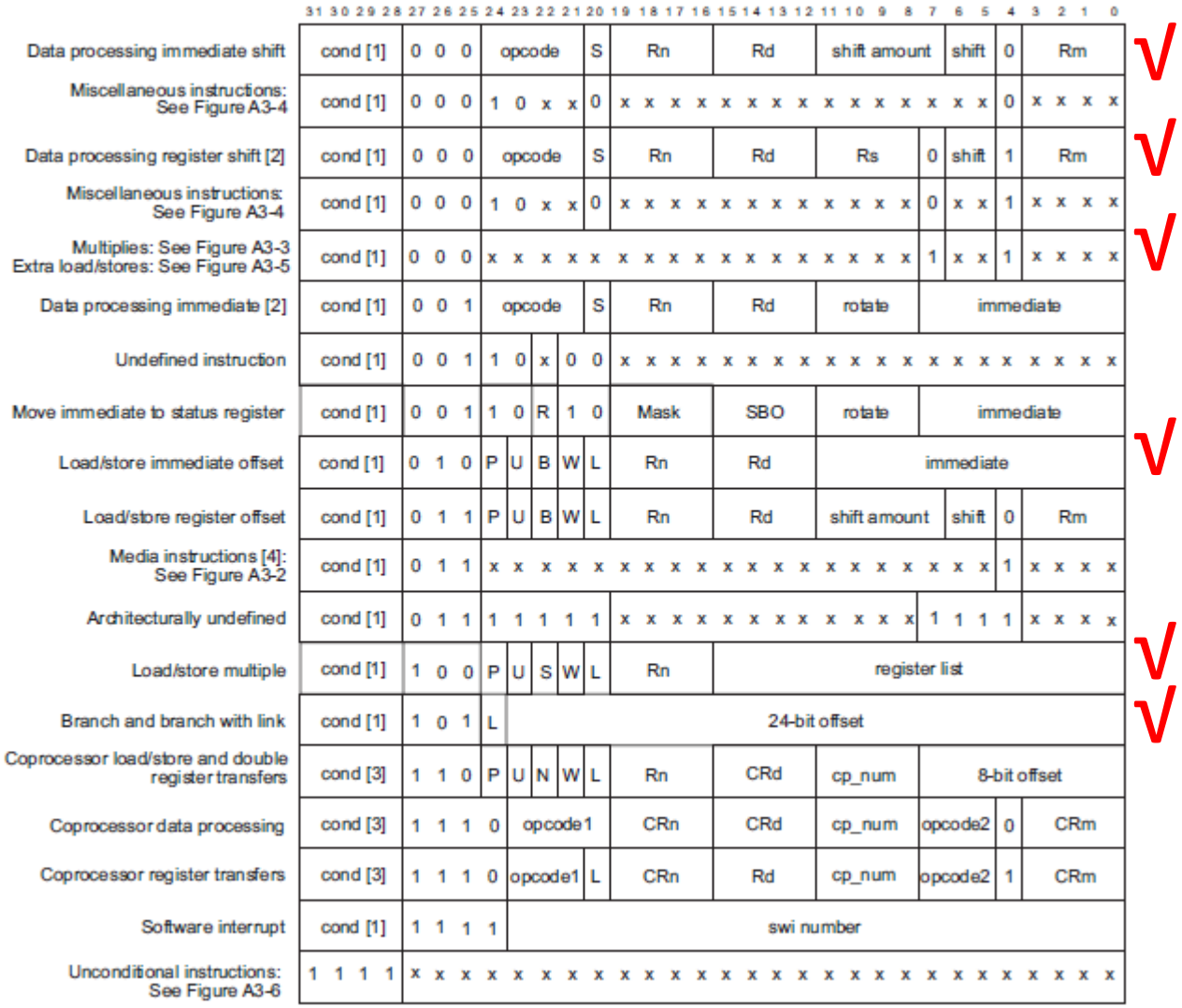
1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)

1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)

1110 = AL - always

1111 = NV - reserved.

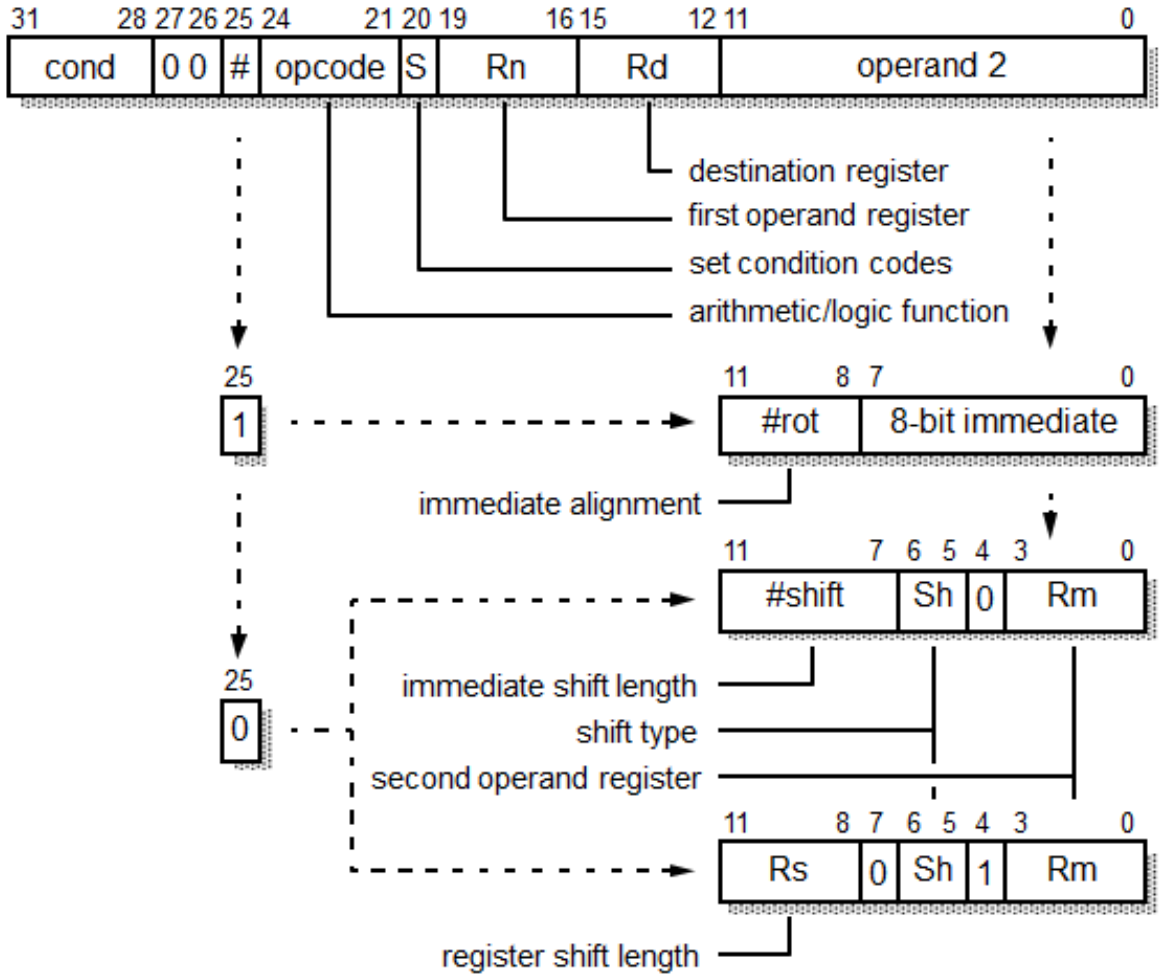
Microprocessor & Computer Architecture (µpCA)



Cond	0	0	I	Opcode				S	Rn				Rd				Operand 2								*Data Processing / PSR Transfer*
Cond	0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm	*Multiply*
Cond	0	0	0	0	1	U	A	S	RdHi				RdLo				Rn				1	0	0	1	Rm	*Multiply Long*
Cond	0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm	*Single Data Swap*
Cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn	*Branch and Exchange*
Cond	0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm	*Halfword Data Transfer: register offset*
Cond	0	0	0	P	U	1	W	L	Rn				Rd				Offset				1	S	H	1	Offset	*Halfword Data Transfer: immediate offset*
Cond	0	1	I	P	U	B	W	L	Rn				Rd				Offset									*Single Data Transfer*
Cond	0	1	1																					1		*Undefined*
Cond	1	0	0	P	U	S	W	L	Rn				Register List													*Block Data Transfer*
Cond	1	0	1	L	Offset																					*Branch*

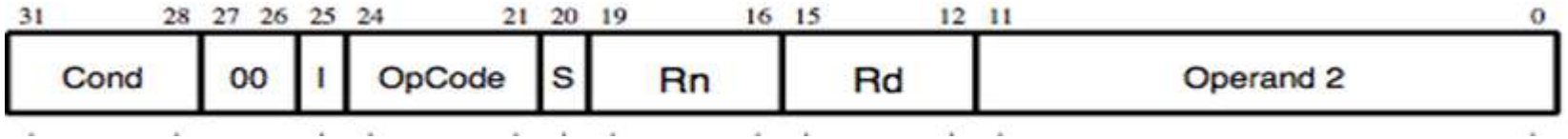
Microprocessor & Computer Architecture (µpCA)

Data Processing Instruction



Opcode [24:21]	Mnemonic
0000	AND
0001	EOR
0010	SUB
0011	RSB
0100	ADD
0101	ADC
0110	SBC
0111	RSC
1000	TST
1001	TEQ
1010	CMP
1011	CMN
1100	ORR
1101	MOV
1110	BIC
1111	MVN

Data Processing Instruction: Example-1



■Ex:

ADD R1, R0, R2

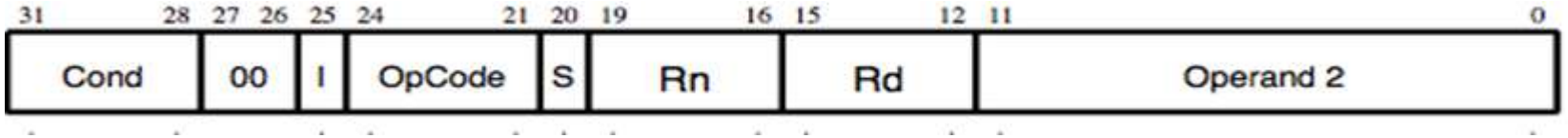
1110 000 0100 S Rn Rd 00000000 Rm

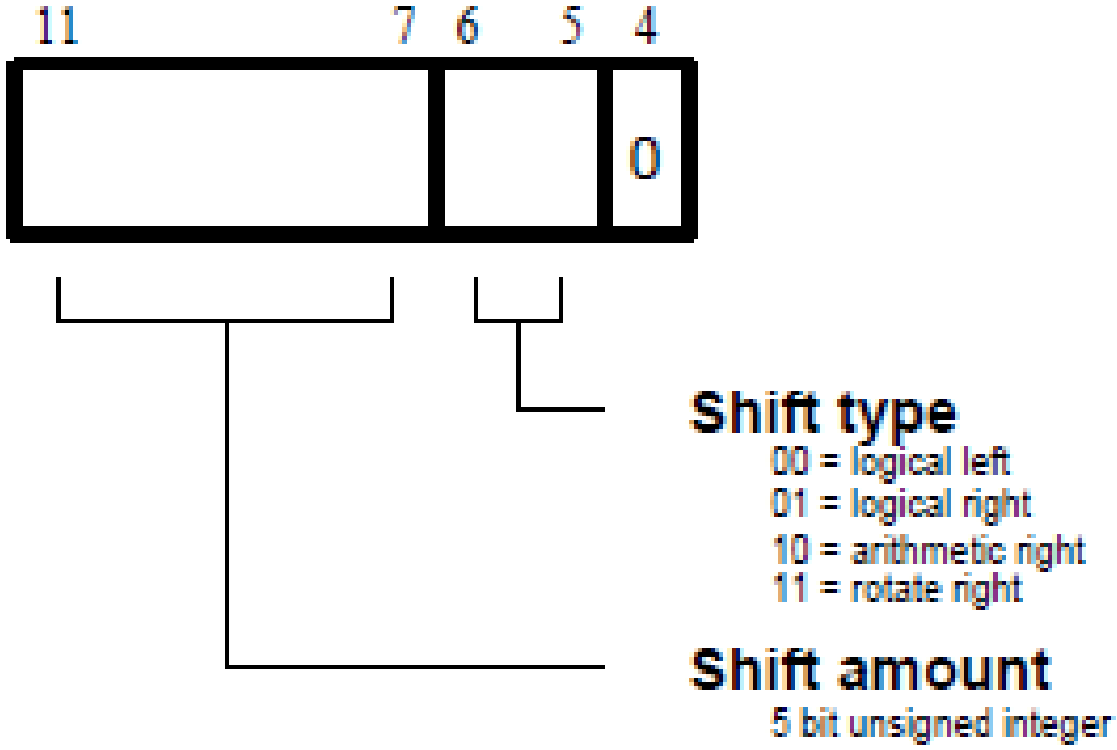
1110 0000 1000 0000 0001 0000 0000 0010

Always ADD R0 R1 R2

E0801002

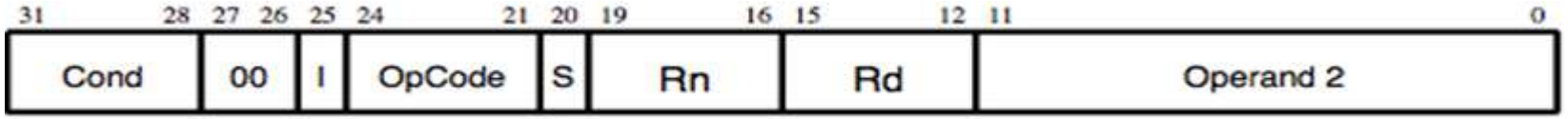
Data Processing Instruction

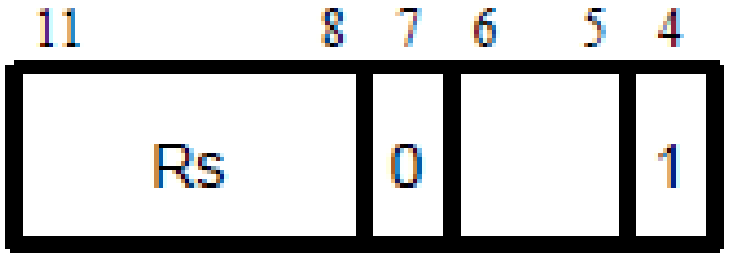




Microprocessor & Computer Architecture (µpCA)

Data Processing Instruction





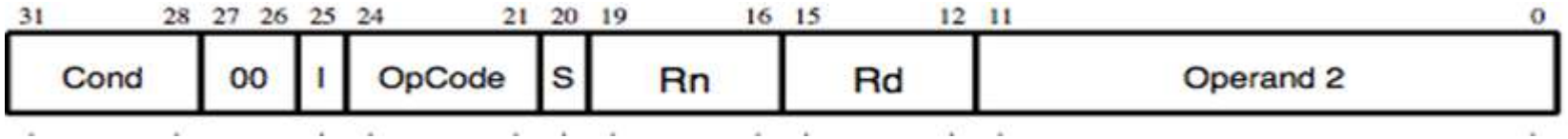
Shift type

00 = logical left
01 = logical right
10 = arithmetic right
11 = rotate right

Shift register

Shift amount specified in bottom byte of Rs

Data Processing Instrucion: Example-2



■Ex:

ADDS R1, R0, R2 LSR #2

1110 000 0100 S Rn Rd Shift Rm

1110 000 0100 S Rn Rd 00010 01 0 Rm

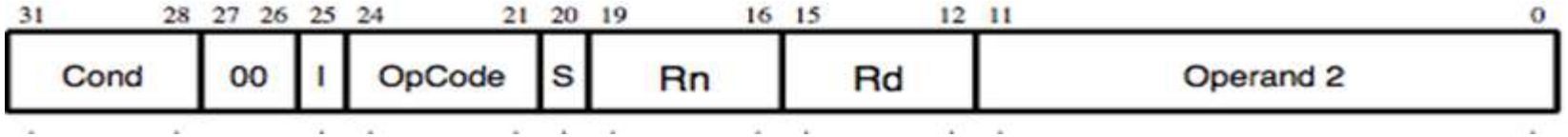
1110 0000 1001 0000 0001 0001 0010 0010

ADDS R0 R1 #2 LSR R2

E0901122

Shift is specified as immediate

Data Processing Instruction: Example-3



■Ex:

ADDS R1, R0, R2 LSR R4

1110 000 0100 S Rn Rd Shift Rm

1110 000 0100 S Rn Rd 0100 0 01 1 Rm

1110 0000 1001 0000 0001 0100 0011 0010

ADDS R0 R1 R4 LSR R2

Shift is specified in Register

E0901432

Next Session

Instruction Encoding

- Branch Instruction
- Multiplication
- Data Transfer Instruction

THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135

Microprocessor & Computer Architecture (µpCA)

UE19CS252

Dr. D. C. Kiran

Department of
Computer Science and Engineering

Microprocessor & Computer Architecture (µpCA)

Instruction Encoding

Dr. D. C. Kiran
Department of Computer Science and Engineering

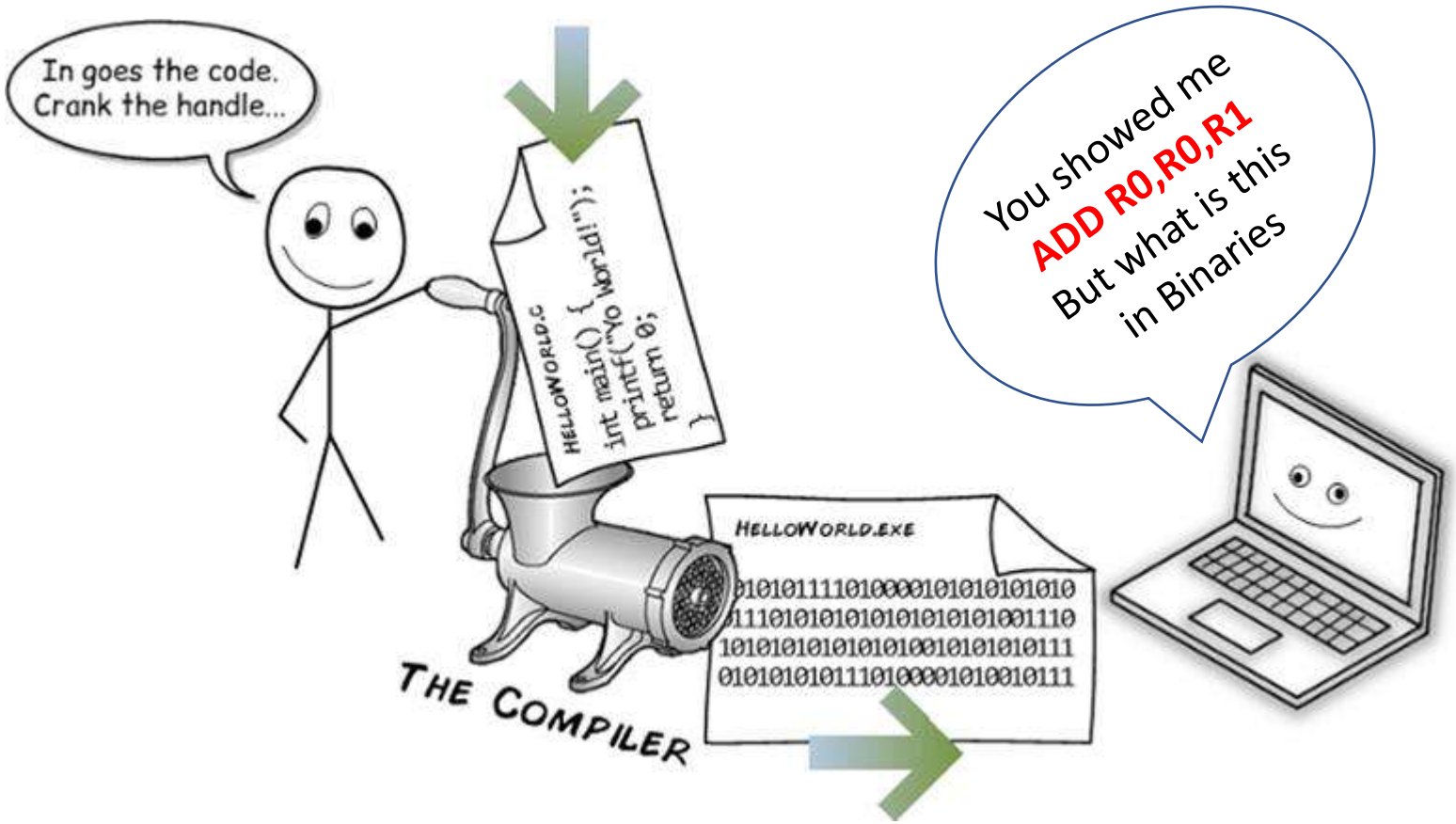
Microprocessor & Computer Architecture (µpCA)

Syllabus

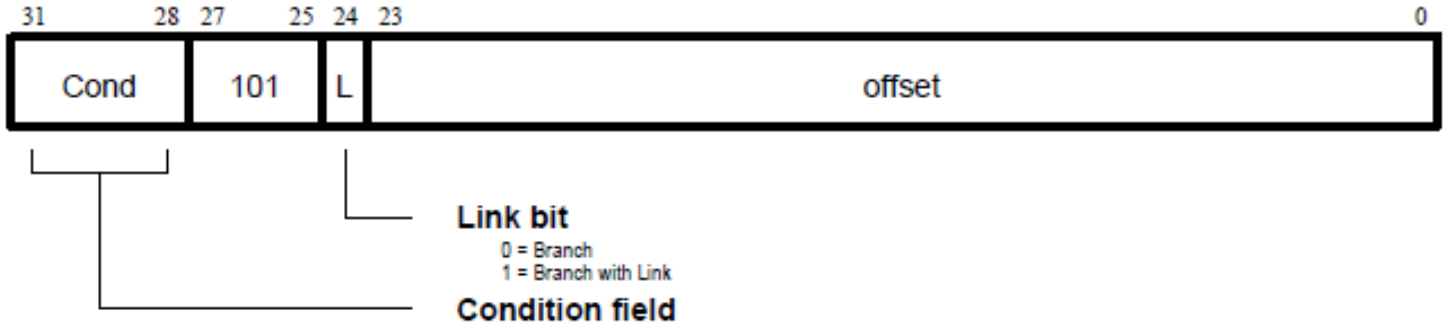
Unit 1: Basic Processor Architecture and Design

- ~~Microprocessor Overview~~
- ~~CISC VS RISC~~
- ~~Introduction to ARM Processor & Applications~~
- ~~ARM Architecture Overview~~
- ~~Different ARM processor Modes~~
- ~~Register Bank~~
- ~~ARM Program structure~~
- ~~ARM Instruction Format~~
- ~~ARM INSTRUCTION SET~~
- **Instruction Encoding**
  - ~~✓ Instruction Layout~~
  - ~~✓ Data Processing Instruction~~
  - ✓ **Branch Instruction**
  - ✓ **Data Transfer Instruction**
  - ✓ **Multiplication Instruction**

Microprocessor & Computer Architecture (µpCA)



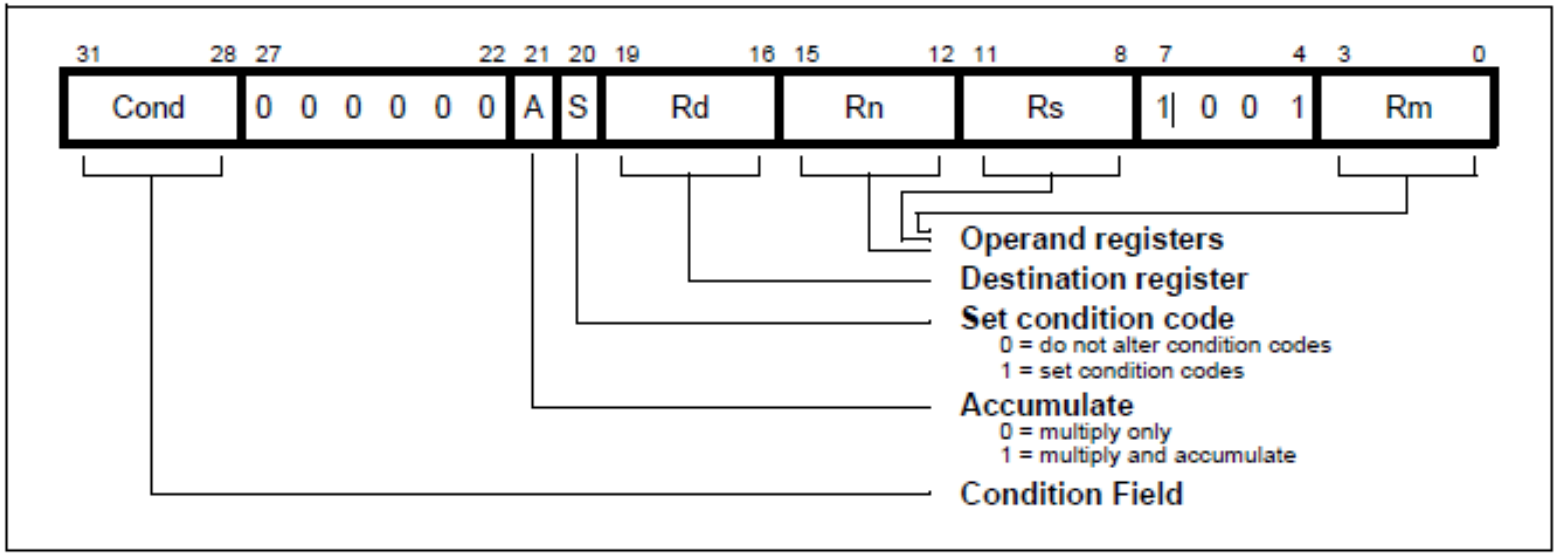
BRANCH INSTRUCTIONS



Ex 1: B LOOP

Ex 2: BL SUBROUTINE

MULTIPLICATION INSTRUCTIONS

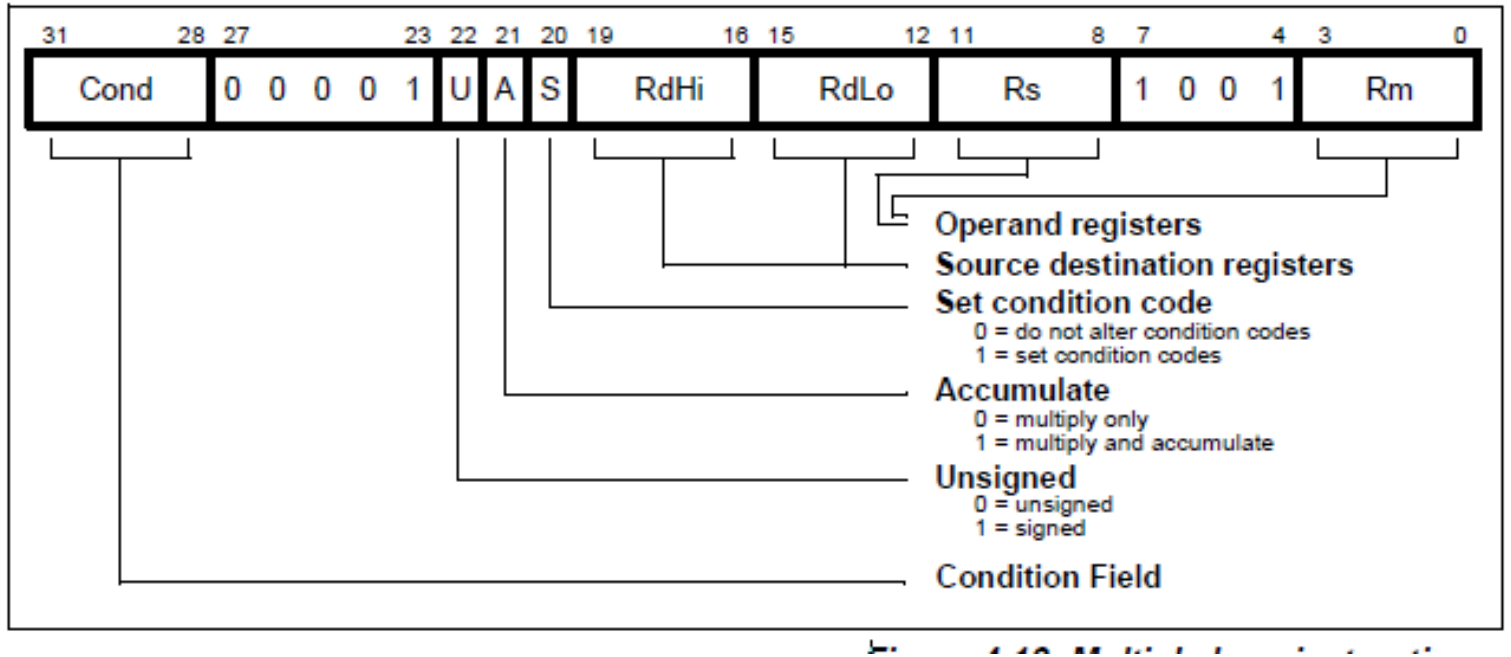


Ex 1: MUL R0, R1, R2

Ex 2: MLA R0, R1, R2 , R3

Microprocessor & Computer Architecture (µpCA)

MULTIPLICATION INSTRUCTIONS

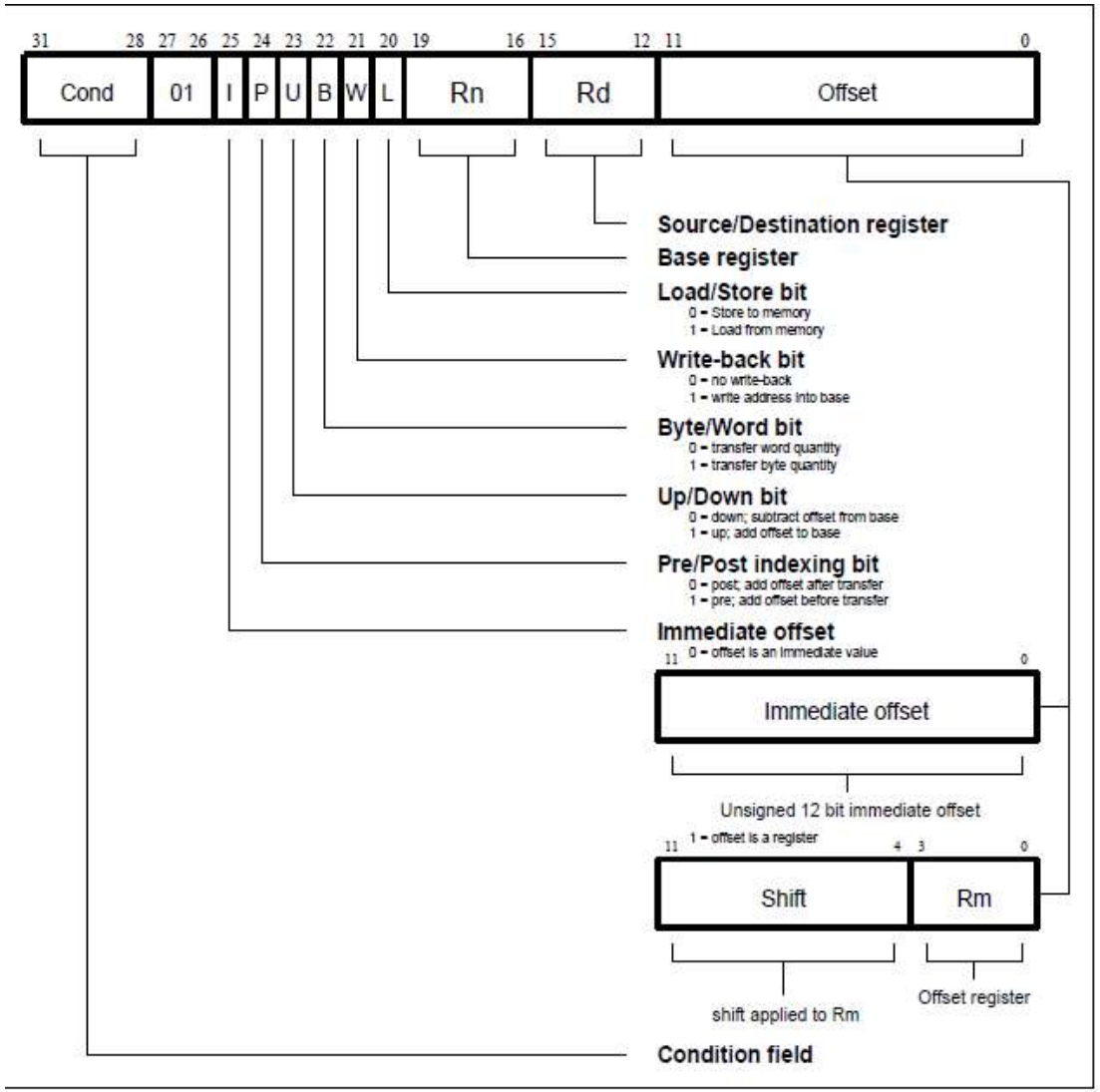


Ex 3: SMULL R0, R1, R2 , R3

Ex 4: UMLAL R0, R1, R2 , R3

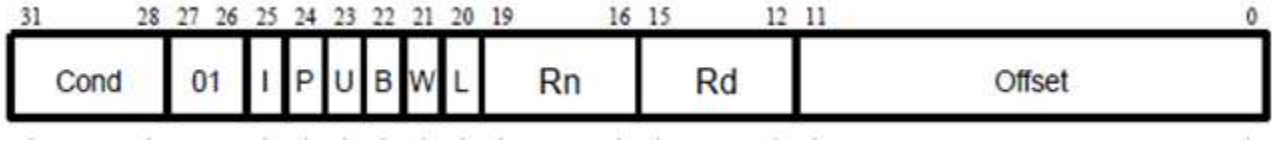
Microprocessor & Computer Architecture (µpCA)

Data Transfer Instruction



Data Transfer Instruction: STR

■Ex:



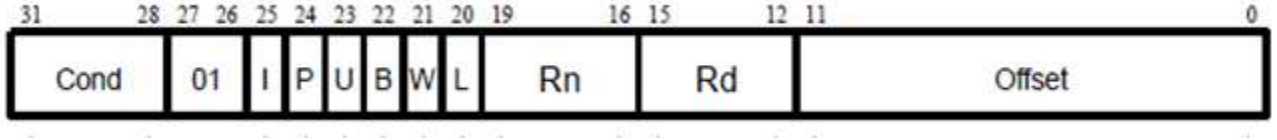
STR R0, [R1]

1110 01 0 0 1 0 1 0 0001 0000 000000000000

1110 01 I P U B W L 0001 0000 000000000000

E4B10000

Data Transfer Instruction: LDR



■ Ex:

LDR R0, [R1], R2

1110 01 1 0 1 0 1 1 0001 0000 00000000 0010

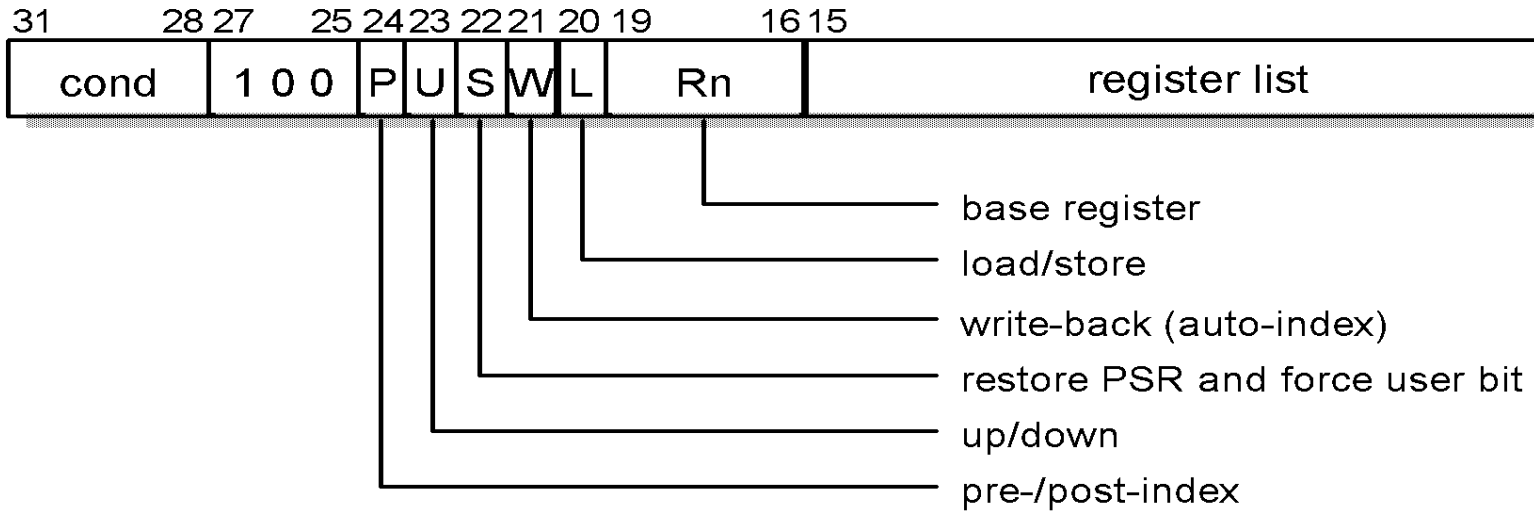
1110 01 I P U B W L 0001 0000 00000000 0010

1110 0110 1011 0001 0000 0000 0000 0010

E6B10002

Microprocessor & Computer Architecture (µpCA)

Block Transfer Instruction



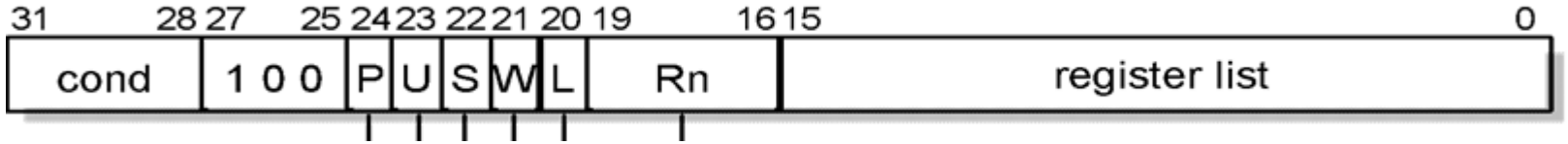
Ex 1: LDMIA R13! , { R0, R5 - R8, R11}

Ex 2: STMIA R13! , { R8, R4- R6, R12}

Block Transfer Instruction: Addressing Mode

Name	Stack	Other	L bit	P bit	U bit
pre-increment load	LDMED	LDMIB	1	1	1
post-increment load	LDMFD	LDMIA	1	0	1
pre-decrement load	LDMEA	LDMDB	1	1	0
post-decrement load	LDMFA	LDMDA	1	0	0
pre-increment store	STMFA	STMIB	0	1	1
post-increment store	STMEA	STMIA	0	0	1
pre-decrement store	STMFD	STMDB	0	1	0
post-decrement store	STMED	STMDA	0	0	0

Block Transfer Instruction: LDM



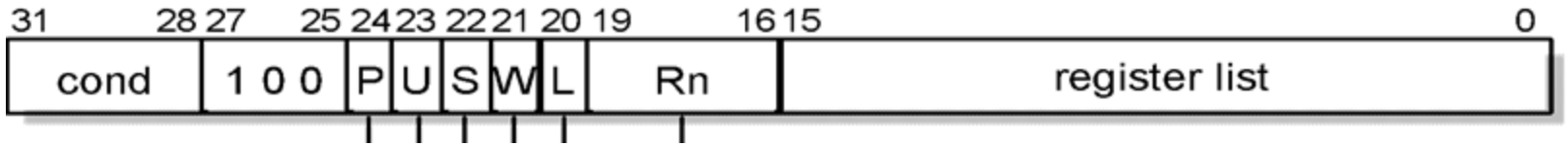
- **LDMIA R13! , { R0, R5 - R8, R11}**

1110 100 **0** 1 **0** **1** **1** 1101 0000 **1** 00**1** **1** **1** **1** 0000**1**

1110 100 **P** **U** **S** **W** **L** 1101 **0000** **R11** **00R8R7R6R5** **0000R0**

E8BD09E1

Block Transfer Instruction: STM



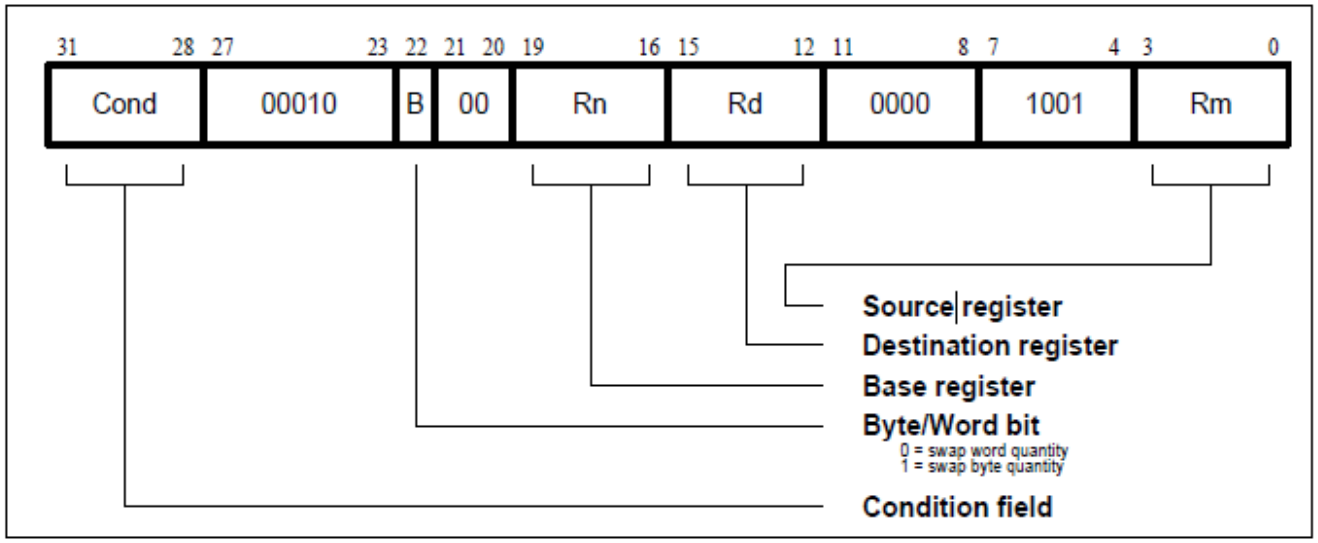
- **STMIB R13! , { R8, R4- R6, R12}**

1110 100 1 0 0 1 0 1101 00010 00101110000

1110 100 P U S W L 1101 000R_{12}0 00R_8 0$R_6R_5R_4$ 0000

E92D1170

SWAP


31 28 27 23 22 21 20 19 16 15 12 11 8 7 4 3 0
Cond
00010
B
00
Rn
Rd
0000
1001
Rm
Source register
Destination register
Base register
Byte/Word bit
0 = swap word quantity
1 = swap byte quantity
Condition field

Next Session

Unit 2
Pipeline Processor

THANK YOU

Dr. D. C. Kiran
Department of Computer Science and Engineering

dckiran@pes.edu

9829935135