18-349: Introduction to Embedded Real-Time Systems
Lecture 3: ARM ASM

Anthony Rowe
Electrical and Computer Engineering
Carnegie Mellon University

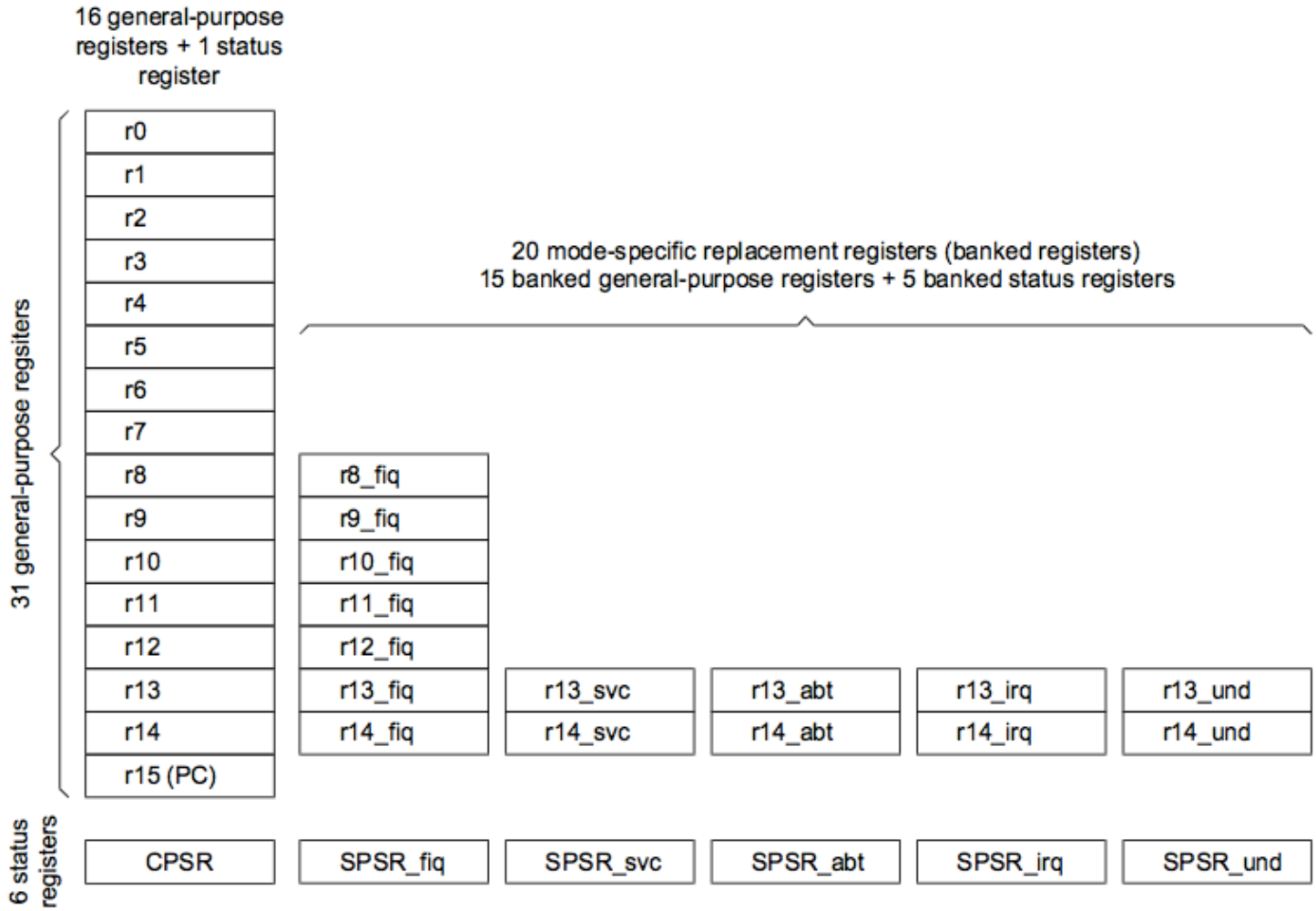
Electrical & Computer ENGINEERING

Carnegie Mellon University

Lecture Overview

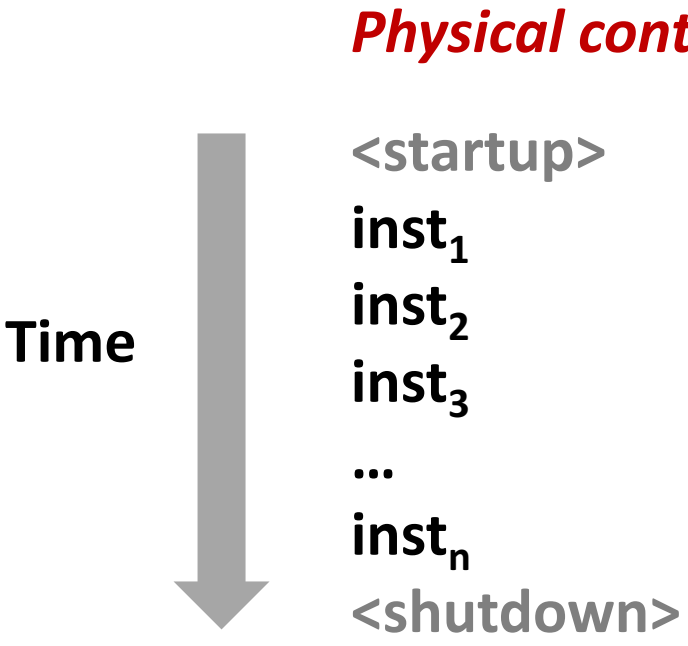
- Exceptions Overview (Review)
- Pipelining
- ARM ASM Introduction
  - Move operations
  - Arithmetic operations
  - Logical operations
  - Comparison operations
  - Multiply operations
  - Conditionals

Reminder: ARM Register Set



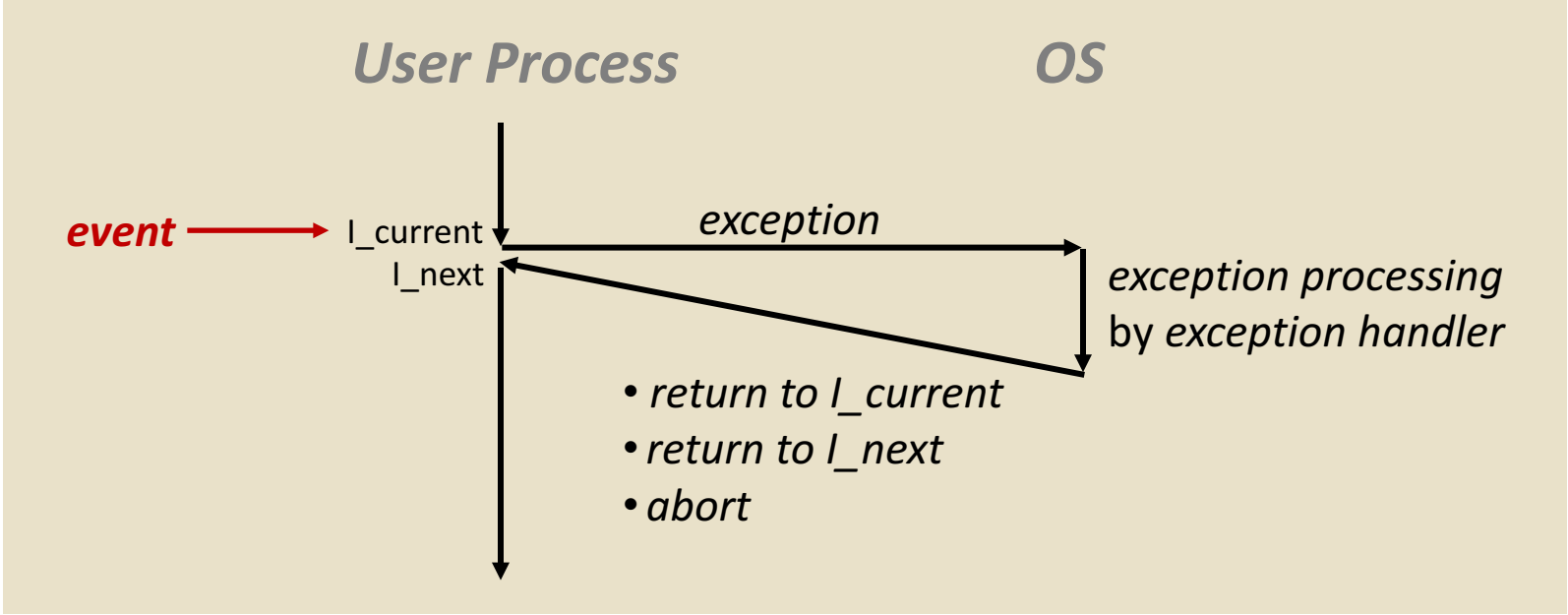
Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU's *control flow* (or *flow of control*)



Exceptions

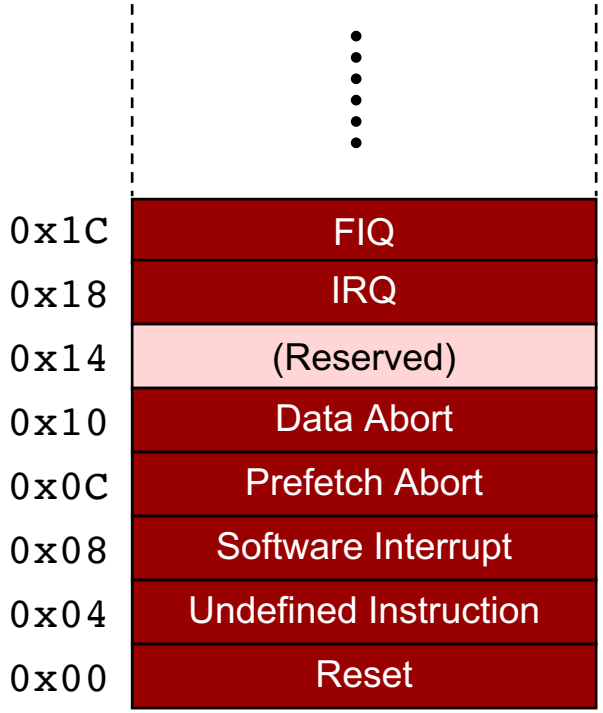
- An *exception* is a transfer of control to supervisory mode in response to some *event* (i.e., change in processor state)



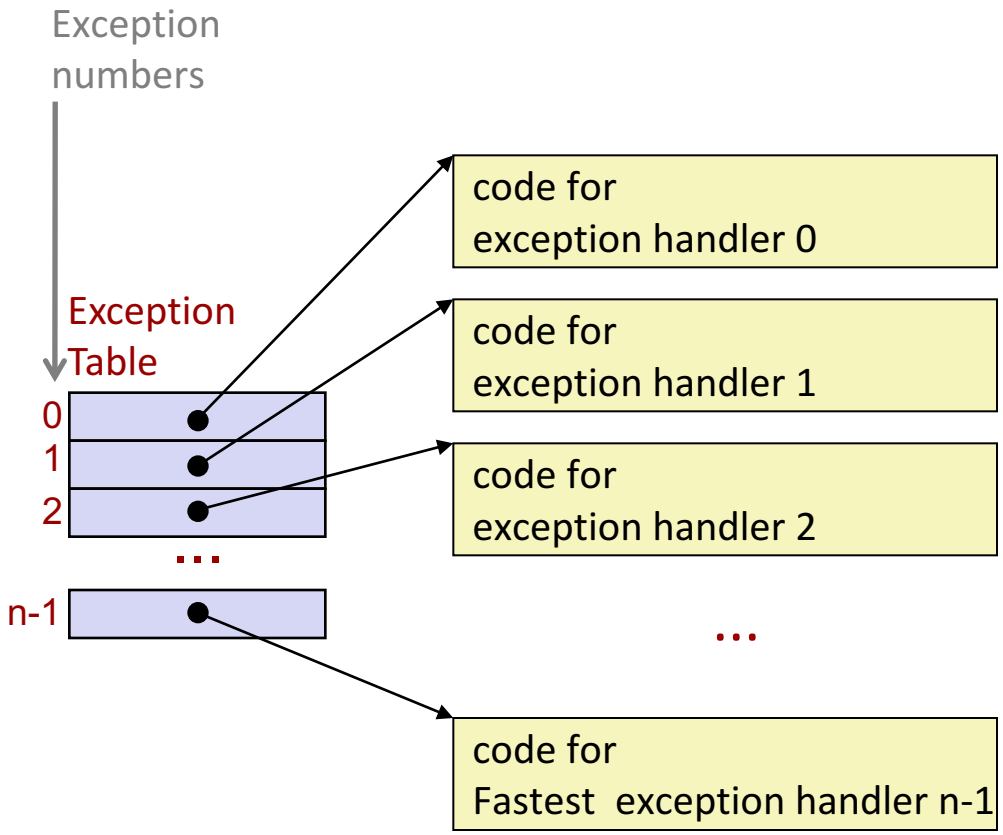
- Examples:
  - div by 0, arithmetic overflow, page fault, I/O request completes, Ctrl-C

The Vector Table

- Reserved area of 32 bytes at the end of the memory map
  - Placed at address 0x0
- One word of space for each exception type
- Contains a Branch or Load `pc` instruction for the exception handler
- Notice that the FIQ exception-handler is at the end of the vector table – why?



The Vector Table

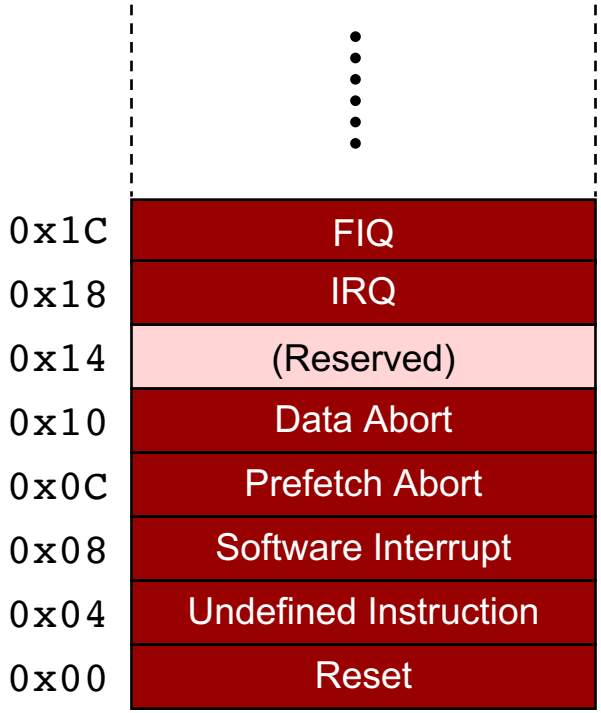


- **Each type of event has a unique exception number k**
- **k = index into exception table (a.k.a. interrupt vector)**
- **Handler k is called each time exception k occurs**

Exception Handling

- When an exception occurs, the ARM:
  - Copies cpsr into spsr_<mode>
  - Sets appropriate cpsr bits
    - Change to ARM state
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the return address in lr_<mode>
  - Sets pc to vector address

- To return, exception handler (the code you write) needs to:
  - Restore cpsr from spsr_<mode>
  - Restore pc from lr_<mode>
  - Handle the general-purpose (gp) registers appropriately



Example: User to Supervisor

- Your program is initially in User mode
- When the processor executes the SWI instruction, the **ARM processor does the following for you automatically:**
  - Copies cpsr into spsr_svc
  - Sets appropriate cpsr mode bits to 10011 (svc mode)
    - Disables IRQs
  - Stores return address (pc – 4) in lr_svc
  - Sets pc to vector address 0x08
- To return, **exception handler (the code you have to write)** needs to:
  - Restore cpsr from spsr_svc (cpsr now has mode bits 10000 = usr mode)
  - Restore pc from lr_svc
- In Supervisor mode, a single instruction can be used to cause the SPSR for the current mode to be copied into CPSR while copying lr into pc
  - MOVS pc, lr

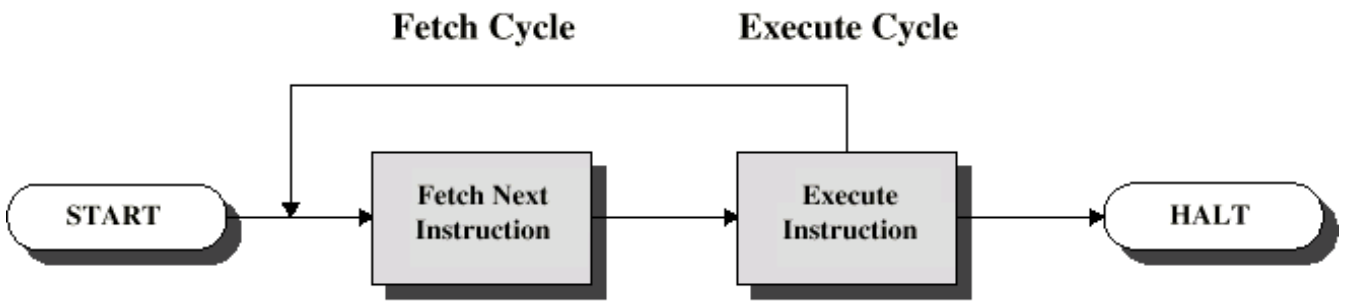
Peak at an Exception Handler

```
; ARM processor has already done its part


S_Handler
  STMFD       sp!, {r0r12, lr}    ; store user's gp registers and lr_svc
  MOV         r1, sp               ; ignore for now
  LDR         r0, [lr, #4]        ; ignore for now
  BIC         r0,r0,#0xff000000    ; ignore for now
  BL          C_SWI_handler        ; ignore for now
  LDMFD       sp!, {r0r12, lr}    ; unstack user's registers and lr_svc
  MOVS        pc, lr               ; return from handler
```

Instruction Cycle

- Two Steps:
  - Fetch
  - Execute



The Fetch Cycle

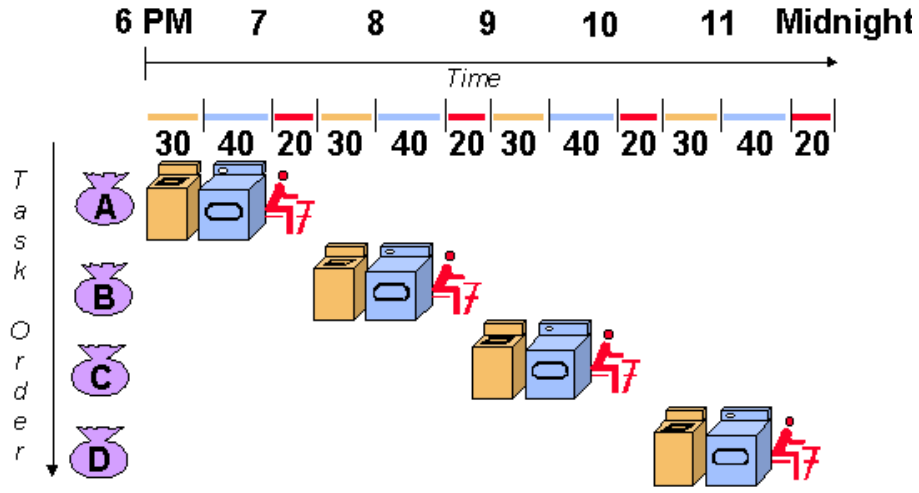
- Program Counter (PC)
  - Holds address of next instruction that the processor should fetch
- So, what happens?
  - Processor fetches instruction from the memory location that the PC points to
  - Processor then increments PC
    - Unless specified otherwise
    - Sometimes, an offset added depending on processor architecture
- Instruction loaded into Instruction Register (IR)
  - Processor interprets instruction and performs required actions

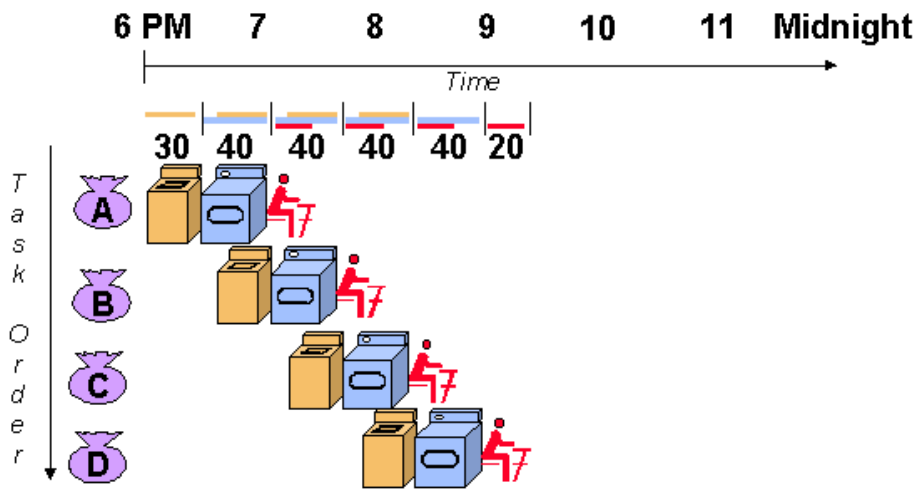
The Execute Cycle

- Processor-memory interactions
  - Data transferred between CPU and main memory
- Processor-I/O interactions
  - Data transferred between CPU and I/O device
- Data processing
  - Some arithmetic or logical operation performed on data
- Control action
  - Alteration of sequence of operations
    - Example: Branch to subroutine
- Combination of the above

Pipelining

- Technique where multiple instructions are overlapped in execution
- Each *stage* completes a part of the execution in parallel
- Pipelining does not decrease/increase the amount of time taken for any single instruction to execute
- Pipelining increases *throughput*, i.e., number of instructions exiting the pipeline in unit time





Source: http://cse.stanford.edu/class/sophomore-college/projects-00/risc/pipelining/

Pipelining Stages

- Processors can vary in the number of pipeline stages

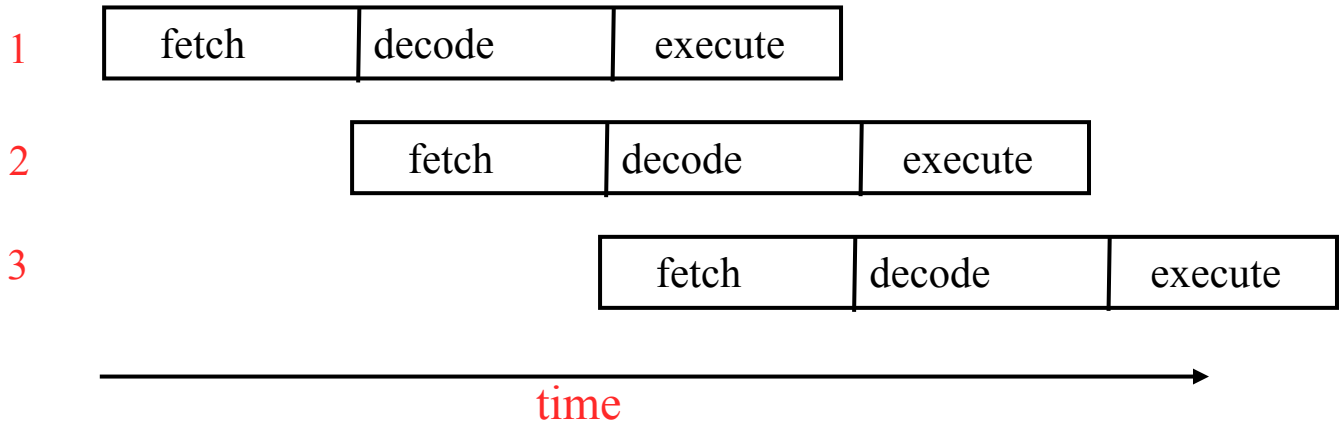
- Often some combination or extension of the following steps
  - Fetch instructions from memory
  - Decode the instruction
  - Execute the instruction or calculate an address
  - Access an operand in data memory
  - Write the result into a register

Issues in Pipelining

- Length of the pipeline depends on the length of the longest step
  - Pipeline rate limited by slowest pipeline stage
- Because RISC instructions are simpler than CISC, they lend themselves better to pipelining
  - RISC instructions are often the same length and can be fetched in one operation (ideally in 1 clock cycle)
- OK, nothing is ever that simple or ideal
  - Data dependencies: Instruction depends on the output value of a previous instruction; value might not yet be ready because that previous instruction is also somewhere (although ahead of the current one) in the pipeline
  - Branch instructions: Next instruction to be executed is decided based on the result of executing another instruction; branch can be conditional on an instruction that has not even made it through the pipeline as yet
  - Pipeline is stalled; empty instructions (*bubbles*) inserted into the pipeline

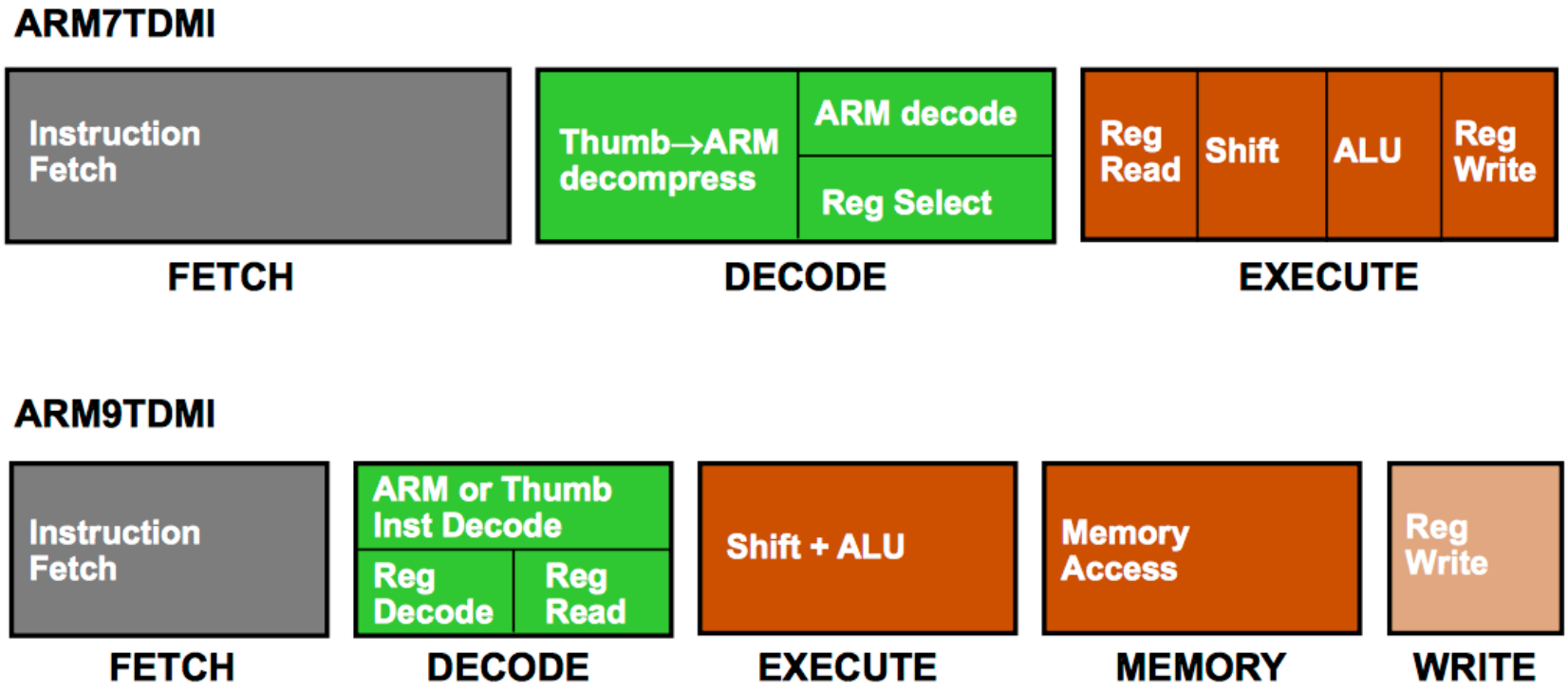
Traditional ARM Pipeline

- ARM traditionally employed a 3-stage pipeline with the following stages
  - Fetch: instruction fetched from memory
  - Decode: instruction is decoded
  - Execute: instruction is executed



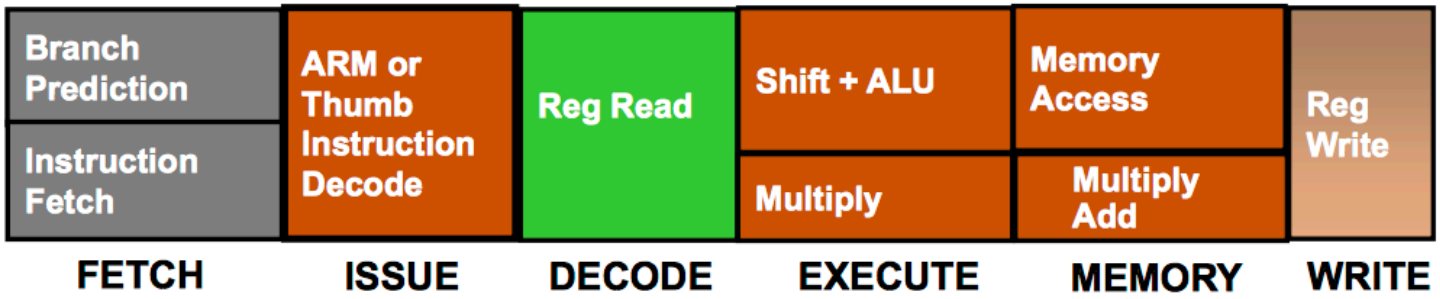
- When processor is executing simple data processing instructions the pipeline enables one instruction to be completed every clock cycle

Pipeline Changes for ARM9TDMI

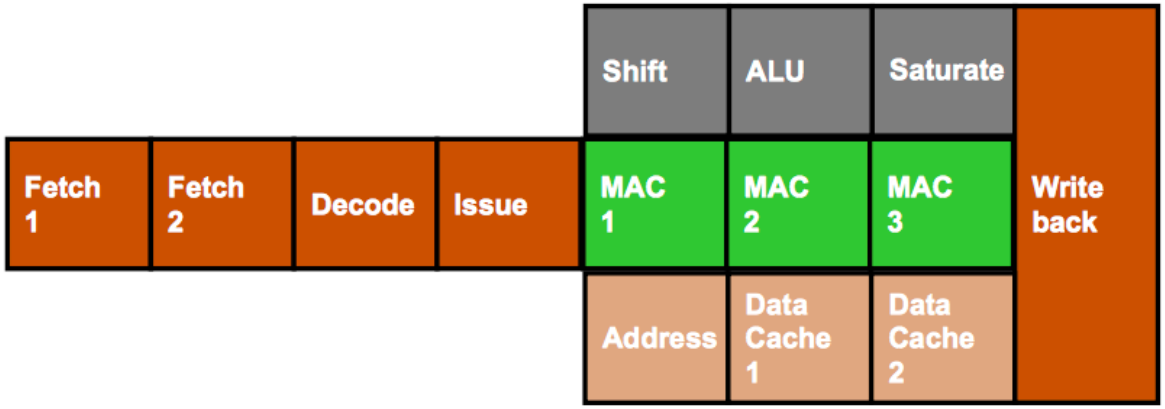


Pipeline changes for ARM10/11

ARM10



ARM11



The ARM Assembly Language

- ARM instructions can be broadly classified as
  - **Data Processing Instructions**: manipulate data within the registers
  - **Branch Instructions**: changes the flow of instructions or call a subroutine
  - **Load-Store Instructions**: transfer data between registers and memory
  - **Software Interrupt Instruction**: causes a software interrupt
  - **Program Status Instructions**: read/write the processor status registers

- All instructions can access r0-r14 directly
- Most instructions also allow use of the pc
- Specific instructions to allow access to cpsr and spsr

ARM Instruction Set Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction Type
Condition				0	0	I	OPCODE				S	Rn				Rs				OPERAND-2												Data processing
Condition				0	0	0	0	0	0	A	S	Rd				Rn				Rs			1	0	0	1	Rm					Multiply
Condition				0	0	0	0	1	U	A	S	Rd HIGH				Rd LOW				Rs			1	0	0	1	Rm					Long Multiply
Condition				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				Swap
Condition				0	1	I	P	U	B	W	L	Rn				Rd				OFFSET												Load/Store - Byte/Word
Condition				1	0	0	P	U	B	W	L	Rn				REGISTER LIST																Load/Store Multiple
Condition				0	0	0	P	U	1	W	L	Rn				Rd				OFFSET 1			1	S	H	1	OFFSET 2					Halfword Transfer Imm Off
Condition				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				Halfword Transfer Reg Off
Condition				1	0	1	L	BRANCH OFFSET																								Branch
Condition				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 Exchange
Condition				1	1	0	P	U	N	W	L	Rn				CRd				CPNum			OFFSET									COPROCESSOR DATA XFER
Condition				1	1	1	0	Op-1				CRn				CRd				CPNum			OP-2			0	CRm				COPROCESSOR DATA OP
Condition								OP-1			L	CRn				Rd				CPNum			OP-2			1	CRm				COPROCESSOR REG XFER
Condition				1	1	1	1	SWI NUMBER																								Software Interrupt

Data Processing Instructions

- Manipulate data within registers
  - Move operations
  - Arithmetic operations
  - Logical operations
  - Comparison operations
  - Multiply operations

- Appending the S suffix for an instruction, e.g, ADDS
  - Signifies that the instruction's execution will update the flags in the cpsr

Typical Data Processing Instruction

- Operation – Specifies the instruction to be performed
- Cond – specify the optional conditional flags which have to be set under which to execute the instruction
  - Almost all ARM instructions can be conditionally executed
- S bit – Signifies that the instruction updates the conditional flags
- Rd – Specifies the destination register
- Rn – Specifies the first source operand register
- ShifterOperand2 – Specifies the second source operand
  - Could be a register, immediate value, or a shifted register/immediate value
- Some data processing instructions may not specify the destination register or the source register

Data Processing Instructions

- Consist of :
  - Arithmetic: **ADD ADC SUB SBC RSB RSC**
  - Logical: **AND ORR EOR BIC**
  - Comparisons: **CMP CMN TST TEQ**
  - Data movement: **MOV MVN**
- These instructions only work on registers, NOT memory

Move Instruction

- `MOV` moves a 32-bit value into a register
- `MVN` moves the NOT of the 32-bit value into a register

PRE

```
r5 = 5
r7 = 8
```

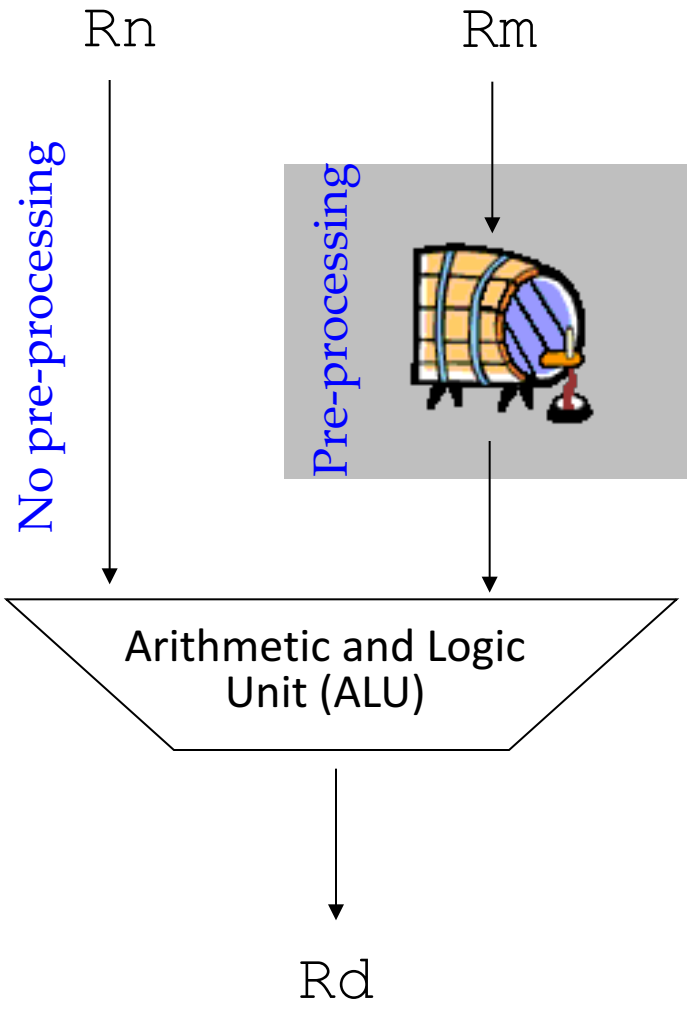
```
MOV r7, r5
```

POST

```
r5 = 5
r7 = 5
```

The ARM Barrel Shifter

- Data processing instructions are processed within the ALU
- ARM can shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before the value enters the ALU
- Can achieve fast multiplies or division by a power of 2
- Data-processing instructions that do not use the barrel shifter
  - MUL (multiply)
  - CLZ (count leading zeros)
  - QADD (signed saturated 32-bit add)



Using the Barrel Shifter

- `LSL` shifts bits to the left, and is similar to the C-language operator <<

PRE

```
r5 = 5
r7 = 8
```

`r7 = r5 * 4 = (r5 << 2)` ←

```
MOV r7, r5, LSL #2
```

POST

```
r5 = 5
r7 =  20
```

Updating the Condition Flags

- The `S` suffix indicates that the `cpsr` should be updated

```
PRE    cpsr = nzcvqiFt_USER
    r0 = 0x00000000
    r1 = 0x80000004

MOVS r0, r1, LSL #1

POST   cpsr = nzCvqiFt_USER
    r0 = 0x00000008
    r1 = 0x80000004
```

Hint: 32-bit version of 0x80000004 is
1000 0000 0000 0000 0000 0000 0000 0100

Arithmetic Operations

- **Operations are:**
  - ADD operand1 + operand2 ; Add
  - ADC operand1 + operand2 + carry ; Add with carry
  - SUB operand1 - operand2 ; Subtract
  - SBC operand1 - operand2 + carry -1 ; Subtract with carry
  - RSB operand2 - operand1 ; Reverse subtract
  - RSC operand2 - operand1 + carry - 1 ; Revers sub with carry

- **Syntax:**
  - <Operation>{<cond>}{S} Rd, Rn, Operand2

- **Examples**
  - ADD r0, r1, r2
  - SUBGT r3, r3, #1
  - RSBLES r4, r5, #5

Arithmetic Instructions

- Subtract

```
PRE     r0 = 0x00000000
        r1 = 0x00000002
        r2 = 0x00000001
SUB r0, r1, r2

POST    r0 = 0x00000001
        r1 = 0x00000002
        r2 = 0x00000001
```

- Addition

```
PRE     r0 = 0x00000000
        r1 = 0x00000005

ADD r0, r1, r1, LSL #1

POST    r0 = 0x0000000f
        r1 = 0x00000005
```

Logical Operations

- **Operations are:**
  - AND operand1 AND operand2 ; and
  - EOR operand1 EOR operand2 ; xor
  - ORR operand1 OR operand2 ; or
  - ORN operand1 NOR operand2 ; or negative (nor)
  - BIC operand1 AND NOT operand2 ; bit clear

- **Syntax:**
  - <Operation>{<cond>}{S} Rd, Rn, Operand2

- **Examples**
  - AND r0, r1, r2
  - BICEQ r2,r3 #7
  - EORS r1,r3,r0

Logical Instructions

Every binary 1 in r2 clears a corresponding bit location in r1

- Bitwise logical operations on two source registers
- AND, ORR, EOR, BIC

Logical OR

```
PRE   r0 = 0x00000000
      r1 = 0x02040608
      r2 = 0x10305070

ORR r0, r1, r2

POST  r0 = 0x12345678
      r1 = 0x02040608
      r2 = 0x10305070
```

Logical bit clear (BIC)

```
PRE   r1 = 0b1111
      r2 = 0b0101

BIC r0, r1, r2

POST  r0 = 0b1010
      r1 = 0b0101
      r2 = 0b0101
```

Multiply Instructions

- Multiple a pair of registers and optionally add (accumulate) the value stored in another register
  - MUL Rd, Rm, Rs Rd = Rm*Rs
  - MLA Rd, Rm, Rs, Rn Rd = Rm*Rs + Rn

- Special instructions called long multiplies accumulate onto a pair of registers representing a 64-bit value
  - SMLAL, SMULL, UMLAL, UMUL

```
PRE   r0 = 0x00000000      MUL r0, r1, r2    POST r0 = 0x04
      r1 = 0x00000002                              r1 = 0x02
      r2 = 0x00000002                              r2 = 0x02
```

Conditional Execution

- Most instruction sets only allow branches to be executed conditionally
- However ARM reuses the condition evaluation hardware to effectively increase number of instructions
  - All instructions have a field to determine if they should be conditionally executed
  - Non-executed instructions consume 1 cycle
- This removes many of the needs for branches, which stall the pipeline (at least 3 refill cycles)

The Condition Field

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction Type
Condition				0	0	I	OPCODE				S	Rn				Rs				OPERAND-2												Data processing

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.

Conditional Mnemonics

Suffix/Mnemonic	Description	Flags tested
EQ	Equal	**Z=1**
NE	Not equal	**Z=0**
CS/HS	Unsigned higher or same	**C=1**
CC/LO	Unsigned lower	**C=0**
MI	Minus	**N=1**
PL	Positive or Zero	**N=0**
VS	Overflow	**V=1**
VC	No overflow	**V=0**
HI	Unsigned higher	**C=1 & Z=0**
LS	Unsigned lower or same	**C=0 or Z=1**
GE	Greater or equal	**N=V**
LT	Less than	**N!=V**
GT	Greater than	**Z=0 & N=V**
LE	Less than or equal	**Z=1 or N=!V**
AL	Always	

Comparison Instructions

The only effect of comparison is to update the condition flag. So now "S bit" needed.

- **Operations are:**
  - **CMP operand1 - operand2 ; Compare**
  - **CMN operand1 + operand2 ; Compare Negative**
  - **TST operand1 ANDoperand2 ; Test**
  - **TEQ operand1 EOR operand2 ; Test equivalence**

- **Syntax:**
  - **<Operation>{<cond>} Rn, Operand2**

- **Examples**
  - **CMP r0, r1**
  - **TSTEQ r2, #5**

Comparison Instructions

- Compare or test a register with a 32-bit value
  - CMN, CMP, TEQ, TST;
- Outcome: Registers under comparison are not affected; cpsr updated
  - Example: CMP x,y sets cpsr flags based on results of x-y (subtract)
  - Example: TST x,y sets cpsr flags based on results of x&y (logical AND)
- Do not need the S suffix

```
PRE    cpsr = nzcvqiFt_USER
    r0 = 4
    r9 = 4
CMP r0, r9

POST   cpsr = nZcvqiFt_USER
    r0 = 4
    r9 = 4
```

Branch Instructions

- To change the flow of execution or to call a routine
- Supports subroutine calls, *if-then-else* structures, loops
- Change of execution forces the pc to point to a new address
- Four different branch instructions on the ARM
  - `B{<cond>} label`
  - `BL{<cond>} label`
  - `BX{<cond>} Rm`
  - `BLX{<cond>} label | Rm`

Value of Conditional

- This improves code density *and* performance by reducing the number of forward branch instructions

```
if (x != 0)
  a = b+c;
else
 a = b-c;
```

```
    CMP   r3,#0
    BEQ   skip
    ADD   r0,r1,r2
    B afterskip
skip
    SUB   r0, r1, r2
afterskip
```

```
CMP   r3,#0
ADDNE r0,r1,r2
SUBEQ r0,r1,r2
```

Examples of Conditional Execution

- Use a sequence of several conditional instructions

```
if (a==0) x=1;
    CMP    r0,#0
    MOVEQ    r1,#1
```

- Set the flags, then use various condition codes

```
if (a==0) x=0;
if (a>0)   x=1;
    CMP    r0,#0
    MOVEQ   r1,#0
    MOVGT    r1,#1
```

- Use conditional compare instructions

```
if (a==4 || a==10) x=0;
    CMP    r0,#4
    CMPNE   r0,#10
    MOVEQ    r1,#0
```

Single Register Data Transfer

- ARM is based on a "load/store" architecture
  - All operands should be in registers
  - Load instructions are used to move data from memory into registers
  - Store instructions are used to move data from registers to memory
  - Flexible – allow transfer of a word or a half-word or a byte to and from memory

```
LDR/STR   Word
LDRB/STRB Byte
LDRH/STRH Halfword
LDRSB     Signed byte load
LDRSH     Signed halfword load
```

- Syntax:
  - `LDR{<cond>}{<size>} Rd, <address>`
  - `STR{<cond>}{<size>} Rd, <address`

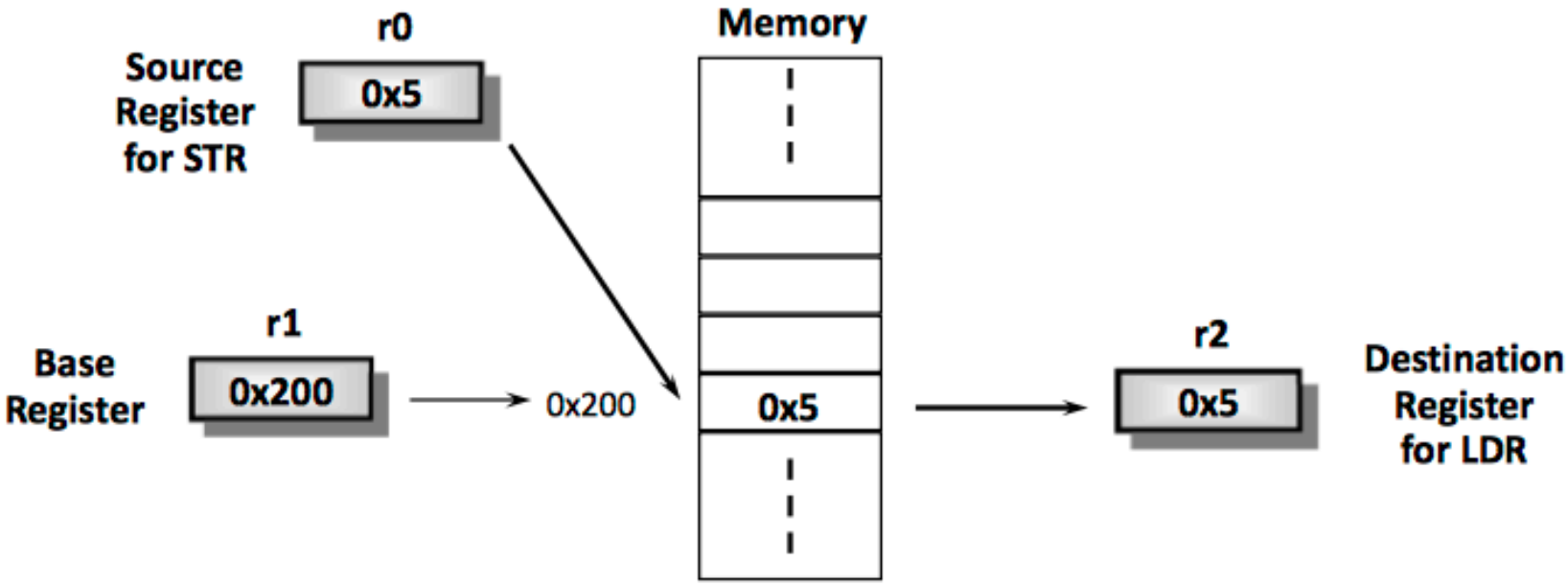
LDR and STR

- LDR and STR instructions can load and store data on a boundary alignment that is the same as the datatype size being loaded or stored
- LDR can only load 32-bit words on a memory address that is a multiple of 4 bytes – 0, 4, 8, and so on
- `LDR r0, [r1]`
  - Loads register r0 with the contents of the memory address pointed to by register r1
- `STR r0, [r1]`
  - Stores the contents of register r0 to the memory address pointed to by register r1
- Register r1 is called the base address register

LDR/STR Example

- The memory location to be accessed is held in a base register

```
STR r0, [r1]      ; Store contents of r0 to location pointed
                  ; to by contents of r1.
LDR r2, [r1]      ; Load r2 with contents of memory location
                  ; pointed to by contents of r1
```



Addressing Modes (1-4)

- ARM provides three addressing modes
  - Preindex with writeback
  - Preindex
  - Postindex
- Preindex mode useful for accessing a single element in a data structure
- Postindex and preindex with writeback useful for traversing an array

Addressing Modes (2-4)

- Preindex
  - Same as preindex with writeback, but does not update the base register
  - Example: `LDR r0, [r1, #4]`
- Preindex with writeback
  - Calculates address from a base register *plus* address offset
  - Updates the address in the base register with the new address
  - The *updated base register value* is the address used to access memory
  - Example: `LDR r0, [r1, #4]!`
- Postindex
  - Only updates the base register *after* the address is used
  - Example: `LDR r0, [r1], #4`

Addressing Modes (3-4)

```
PRE    r0 = 0x00000000
       r1 = 0x00009000
       mem32[0x00009000] = 0x01010101
       mem32[0x00009004] = 0x02020202
```

Preindexing with writeback

```
LDR r0, [r1, #4]!

POST   r0 = 0x02020202
       r1 = 0x00009004
```

Preindexing

```
LDR r0, [r1, #4]

POST r0 = 0x02020202
     r1 = 0x00009000
```

Postindexing

```
LDR r0, [r1], #4

POST   r0 = 0x01010101
       r1 = 0x00009004
```

Addressing Modes (4-4)

- Address `<address>` accessed by LDR/STR is specified by
  - A base register plus an offset

- Offset takes one of the three formats
  1. Immediate: offset is a number that can be added to or subtracted from the base register

     Example:
     ```
     LDR r0,[r1,  #8];      r0 ⇦ mem[r1+8]
     LDR r0,[r1, #-8];      r0 ⇦ mem[r1-8]
     ```

  2. Register: offset is a general-purpose register that can be added to or subtracted from the base register

     Example:
     ```
     LDR r0,[r1,  r2];      r0 ⇦ mem[r1+r2]
     LDR r0,[r1, -r2];      r0 ⇦ mem[r1-r2]
     ```

  3. Scaled Register: offset is a general-purpose register shifted by an immediate value and then added to or subtracted from the base register

     Example:
     ```
     LDR r0,[r1,r2, LSL #2];      r0 ⇦ mem[r1+4*r2]
     ```

Summary

- Exceptions
  - Vector Table
- Pipelining
  - What is it?
  - Why do we do it?
- ARM ISA Introduction
- **Next Lecture**
  - Addressing Modes (reviewed) and Block Data Transfer
  - Stack
  - Memory Mapped IO