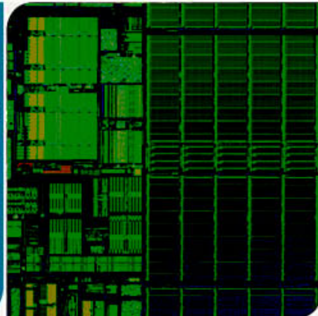
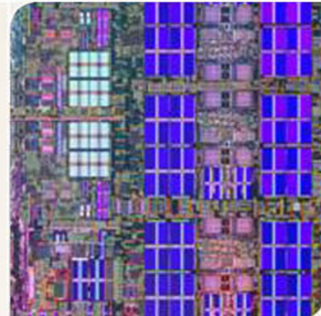
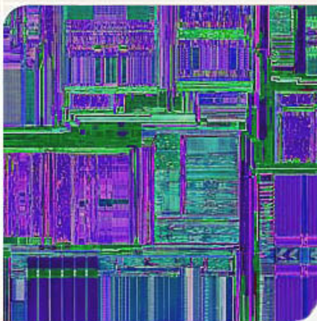
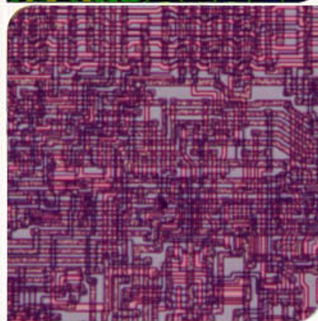
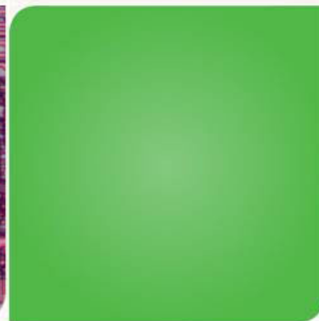
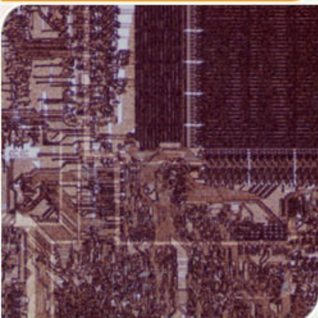






CS4803/CS8803 PGC Design and Programming of Game Console

Spring 2012

Prof. Hyesoon Kim



Georgia Tech

College of Computing

ARM Architecture version summary

- ARM v7
  - Cortex- A, Cortex-R, Cortex-M3, Qualcomm Scorpion
- ARM v6
  - Cortex-M0, Cortex-M1
- ARM v5
  - ARM7, ARM9, ARM10, StrongARM, Intel XScale

Instruction Types

- processing instructions
- transfer instructions
- instructions

Data Processing Instructions

31 – 28	27 26	25	24 – 21	20	19 – 16	15 – 12	11 – 0
cond	0 0	#	opcode	S	Rn	Rd	operand 2

S → Set condition codes

= 1:

11 – 8	7 – 0
#rot	8-bit immediate

= 0:

11 – 7	6 – 5	4	3 – 0
#shift	sh	0	Rm

11 – 8	7	6 – 5	4	3 – 0
Rs	0	sh	1	Rm

Register shift length

Shift type

2nd operand

Immediate Operands

- Using 12 bits how to represent 32-bit immediate value?
-

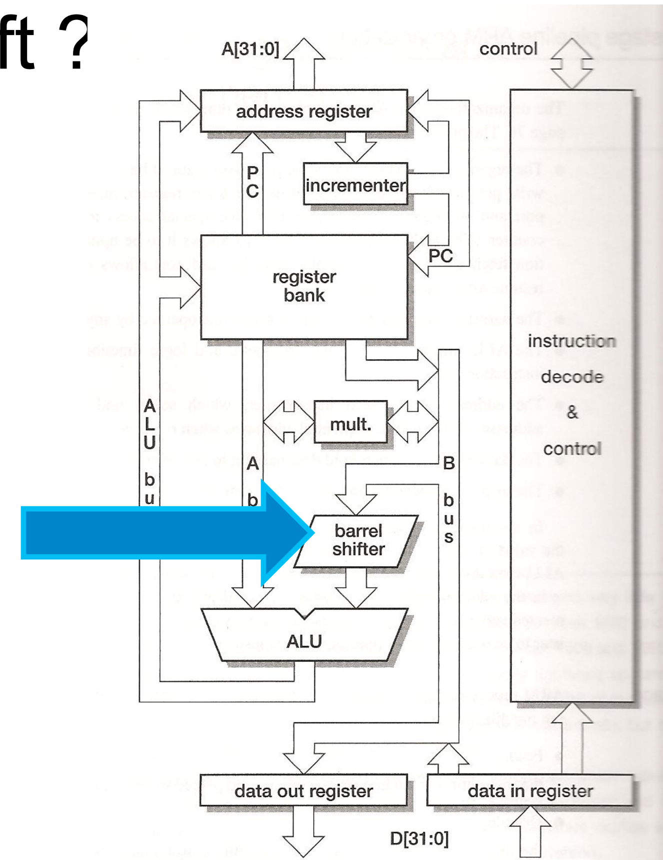
Georgia Tech College of Computing

Shifted Register Operations

- ADD r3, r2, r1, LSL #3; r3:=r2+r1*8
- Logical shift vs. Arithmetic shift ?
  - E.g.) b1011 , Carry:1
  - LSL, 1 : b0110
  - LSR, 1: b0101
  - ASL, 1: b0110
  - ASR, 1: b1101
  - ROR, 1: b1101
  - RRX, 1: b1101 carry: 1

Coming from carry bit

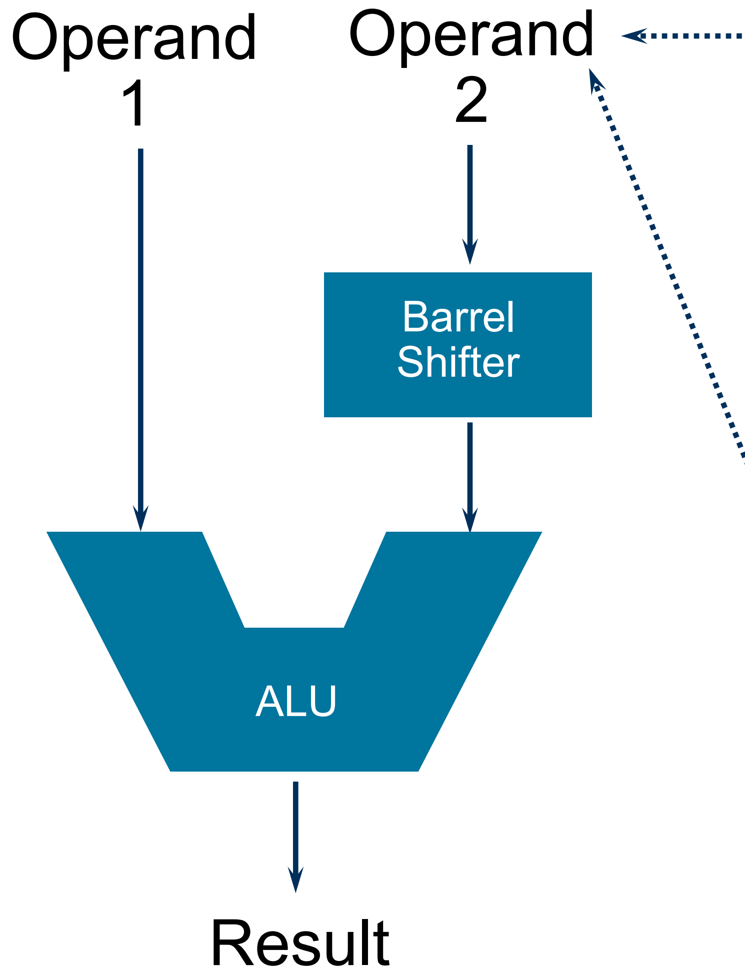
Input to the ALU



- Use register to specify shift
- ADD r5,r5,r3, LSL r2; r5 := r5+r3 x 2^(r2)

Steve Furber, ARM system-on-chip architecture 2nd edition

Using the Barrel Shifter: The Second Operand

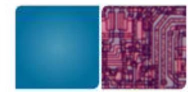


Register, optionally with shift operation

- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.
- Used for multiplication by constant

Immediate value

- 8 bit number, with a range of 0-255.
  - Rotated right through even number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers



Exercise

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.
- Syntax:

```
<Operation>{<cond>}{S} Rd, Rn, Operand2
```

  - Comparisons set flags only - they do not specify Rd
  - Data movement does not specify Rn
- Second operand is sent to the ALU via barrel shifter.

ARM Data Processing Instructions

Opcode [24:21]	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd := Rn AND Op2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn EOR Op2
0010	SUB	Subtract	Rd := Rn – Op2
0011	RSB	Reverse subtract	Rd: = Op2 – Rn
0100	ADD	Add	Rd: = Rn + Op2
0101	ADC	Add with carry	Rd: = Rn + Op2 +C
0110	SBC	Subtract with carry	Rd: = Rn – Op2 + C-1
0111	RSC	Reverse subtract with carry	Rd: = Op2- Rn+C-1
1000	TST	Test	Scc on Rn AND Op2
1001	TEQ	Test equivalence	Scc on Rn EOR Op2
1010	CMP	Compare	Scc on Rn - Op2
1011	CMN	Compare negated	Scc on Rn + Op2
1100	ORR	Logical bit-wise Or	Rd: =Rn OR Op2
1101	MOV	Move	Rd: = Op2
1110	BIC	Bit clear	Rd: =Rn AND NOT Op2
1111	MVN	Move negated	Rd: = NOT Op2

Steve Furber, ARM system-on-chip architecture 2nd edition

Condition Code Set

- S bit (bit 20)
  - 1: condition code is set
  - 0: condition code is unchanged
- N: 1: result is negative 0: result is 0 or positive
  - N = result [31]
- Z: 1: zero 0: non-zero
- C: Carry-out from ALU when the operation is arithmetic
  - ADD, ADC, SUB, SBC, RSB, CMP, CMN
  - Carry out from shifter
- V: overflow , non-arithmetic operations do not touch V-bit
  - Only for signed operations

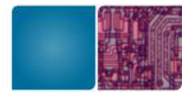
Condition Codes

- The possible condition codes are listed below:
  - Note AL is the default and does not need to be specified

Suffix	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	

Steve Furber, ARM system-on-chip architecture 2nd edition

ADD vs. ADDC



Example of Condition Code Execution

- Use a sequence of several conditional instructions

```
if (a==0) { a = 1; func(1); }
    CMP      r0,#0 // a == 0
    MOVEQ    r0,#1  // a = 1
    BLEQ     func   // func
```

- 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
```


Georgia Tech College of Computing

Multiply

- Syntax:
  - MUL{<cond>}{S} Rd, Rm, Rs   Rd = Rm * Rs
  - MLA{<cond>}{S} Rd,Rm,Rs,Rn   Rd = (Rm * Rs) + Rn
  - [U|S]MULL{<cond>}{S} RdLo, RdHi, Rm, Rs   RdHi,RdLo := Rm*Rs
  - [U|S]MLAL{<cond>}{S} RdLo, RdHi, Rm, Rs   RdHi,RdLo := (Rm*Rs) +RdHi,RdLo
- Cycle time
  - Basic MUL instruction
    - 2-5 cycles on ARM7TDMI
    - 1-3 cycles on StrongARM/XScale
    - 2 cycles on ARM9E/ARM102xE
  - +1 cycle for ARM9TDMI (over ARM7TDMI)
  - +1 cycle for accumulate (not on 9E though result delay is one cycle longer)
  - +1 cycle for “long”
- Above are “general rules” – timing can be vary by architecture

Multiply Instructions

Opcode [23:21]	Mnemonic	Meaning	Effect
000	MUL	Multiply (32-bit result)	Rd := (Rm*Rs)[31:0]
001	MLA	Multiply-accumulates (32-bit result)	Rd := (Rm*Rs+Rn)[31:0]
100	UMULL	Unsigned multiply long	RdHi:RdLo:=Rm*Rs
101	UMLAL	Unsigned multiply-accumulate long	RdHi:RdLo:+=Rm*Rs
110	SMULL	Signed multiply long	RdHi:RdLo:=Rm*Rs
111	SMLAL	Signed multiply-accumulate long	RdHi:RdLo:+=Rm*Rs

- RdHi:RdLo: 64-bit format RdHi: MSB 32 bits, RdLo: LSB 32 bits
- N: Rd[31] or RdHi[31]
- Z: Rd or RdHi and RdLo are Zero
- C: meaningless
- V: unchanged
- Early ARM supports only 32 bits Multiply operations. 64 bit multiply instructions are supported from ARM7.

Data Transfer Instructions

- Data transfer between registers and memory.
- Single word and unsigned byte data transfer instructions
- Half-word and signed byte data transfer instructions
- Multiple register transfer instructions
  - Copy subset or multiple registers to memory
- Swap memory and register instructions (SWP)
- Status register to general register transfer instructions

Single register data transfer

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

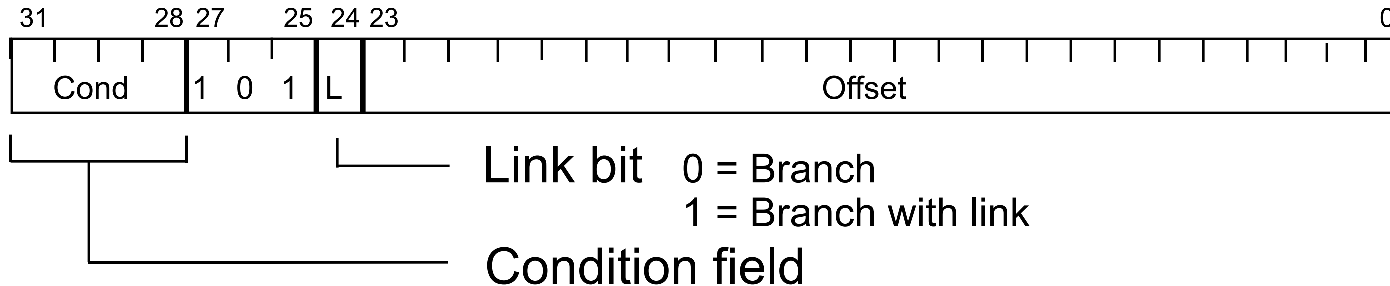
- Memory system must support all access sizes

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

e.g. `LDREQB`

Branch instructions

- Branch : `B{<cond>} label`
- Branch with Link : `BL{<cond>} subroutine_label`



- The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC
  - ± 32 Mbyte range
  - How to perform longer branches?

Branch Code and Predicated Code

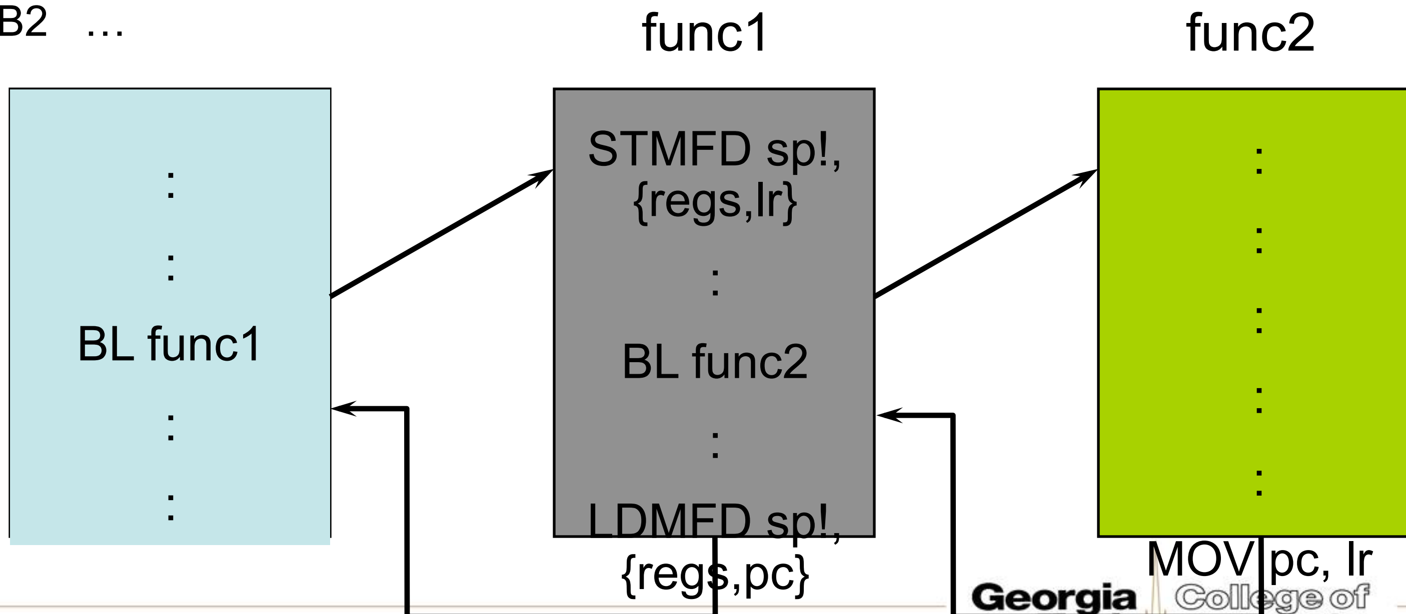
- Pros and Cons

Exercise

Nested Sub-routine Calls

- Nested sub-routine calls
- Link register (r14) needs to be stored

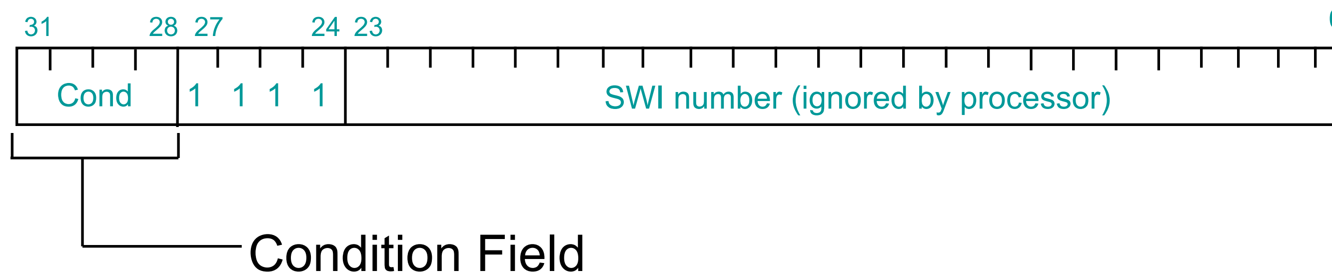
```
        BL   SUB1
         ....
SUB1   STMFD   r13!,   { r0-r2, r14} ; save work and link regs
        BL   SUB2

SUB2   ...
```



Stack Addressing

- Ascending stack
- Descending stack
- Full stack
- Empty stack

Software Interrupt (SWI)



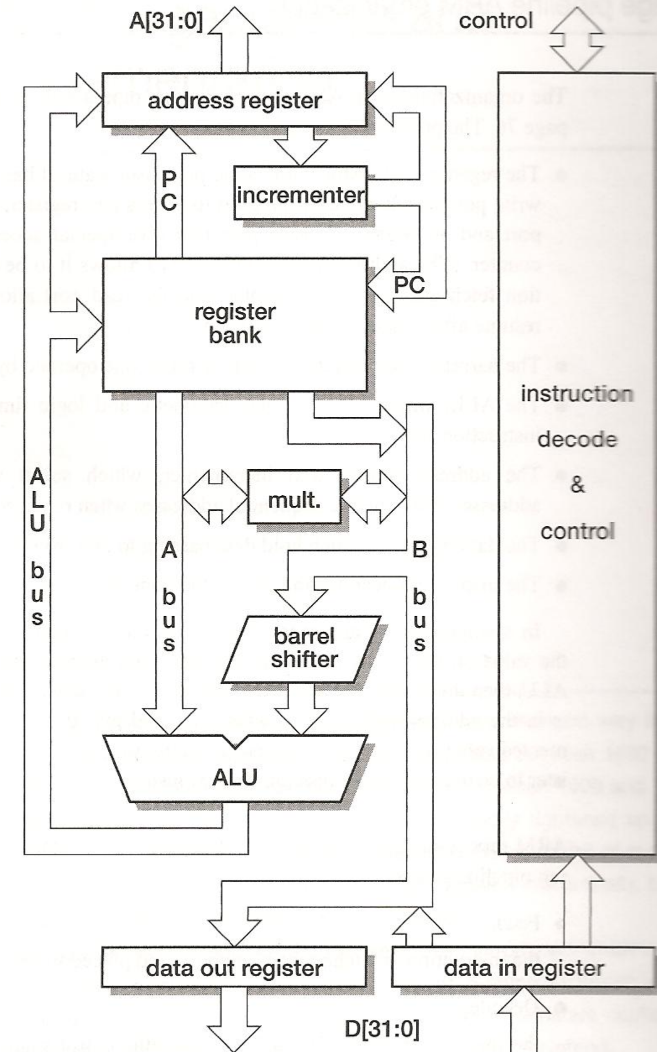
- Causes an exception trap to the SWI hardware vector
- The SWI handler can examine the SWI number to decide what operation has been requested.
- By using the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.
- Syntax:
  - `SWI{<cond>} <SWI number>`

Memory Indexing using LDR/STR

- Address accessed by LDR/STR is specified by a base register plus an offset
- Register indirect memory addressing
  - LDR r0, [r1] ; r0 := mem_{32}[r1]
  - STR r0, [r1] ; mem_{32}[r1] := r0
- Particular location:
  - Set base register
    - an address within 4K bytes of the location
- Base plus offset addressing
  - LDR r2, [r1, #4] ; r2 := mem_{32}[r1+4]
  - LDR r2, [r1, #4]! ; r2 := mem_{32}[r1+4]; r1: = r1+4
  - ! Indicates update the base register (auto update)
- Post-indexed register
  - LDR r2, [r1], #4 ; r2 := mem_{32}[r1]; r1: = r1+4

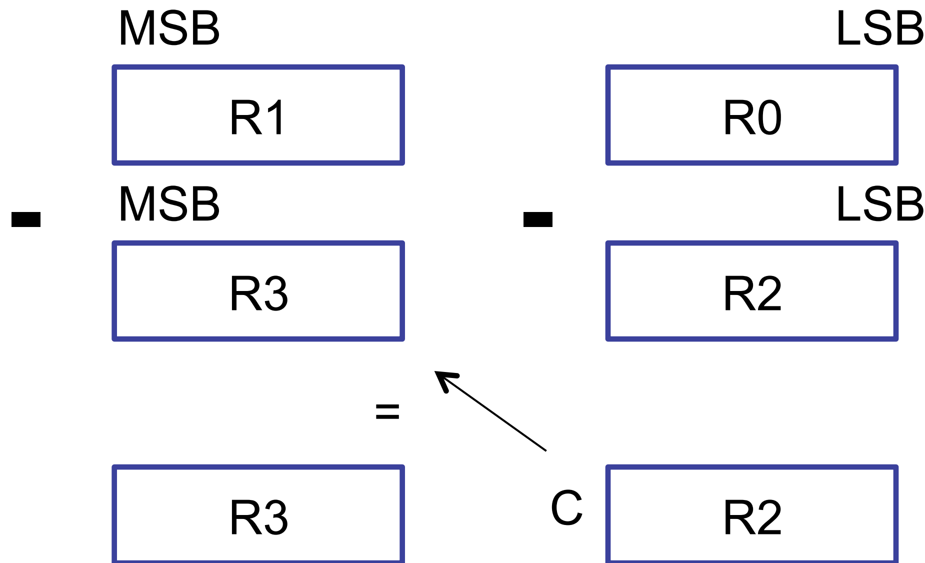
Auto-updating Why?

- ARM: RISC type operations
- Auto-updating: CISC style
- Why?



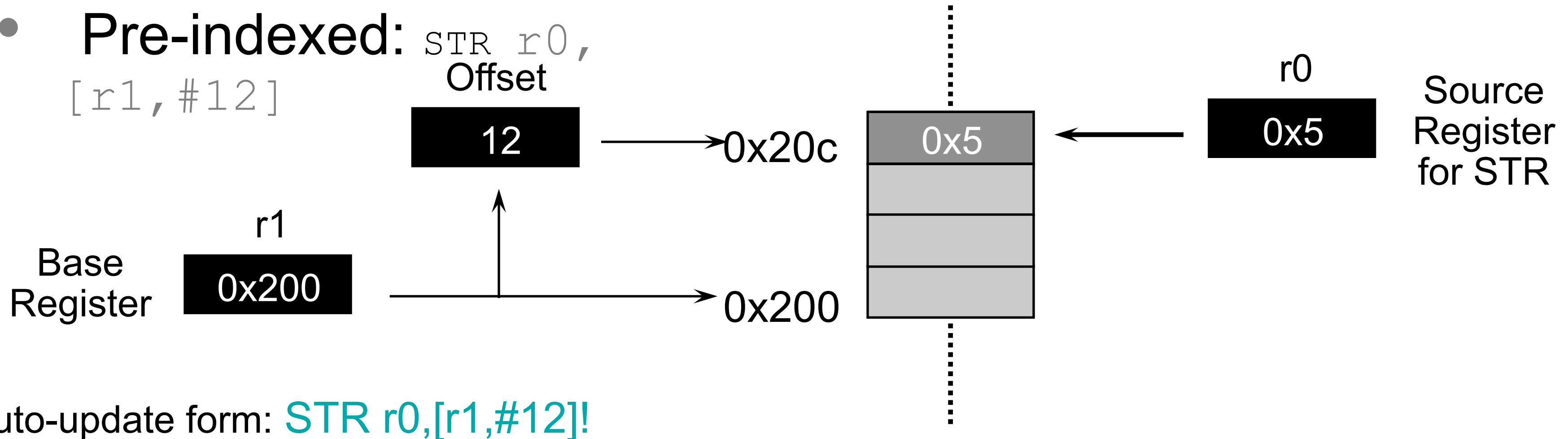
Design 64-bit Subtraction using 32-bit registers

- SUBS R2, R0, R2 := R2 = R0- R2
- SUBC R3, R1, R3 := R3 = R1-R3 +C -1



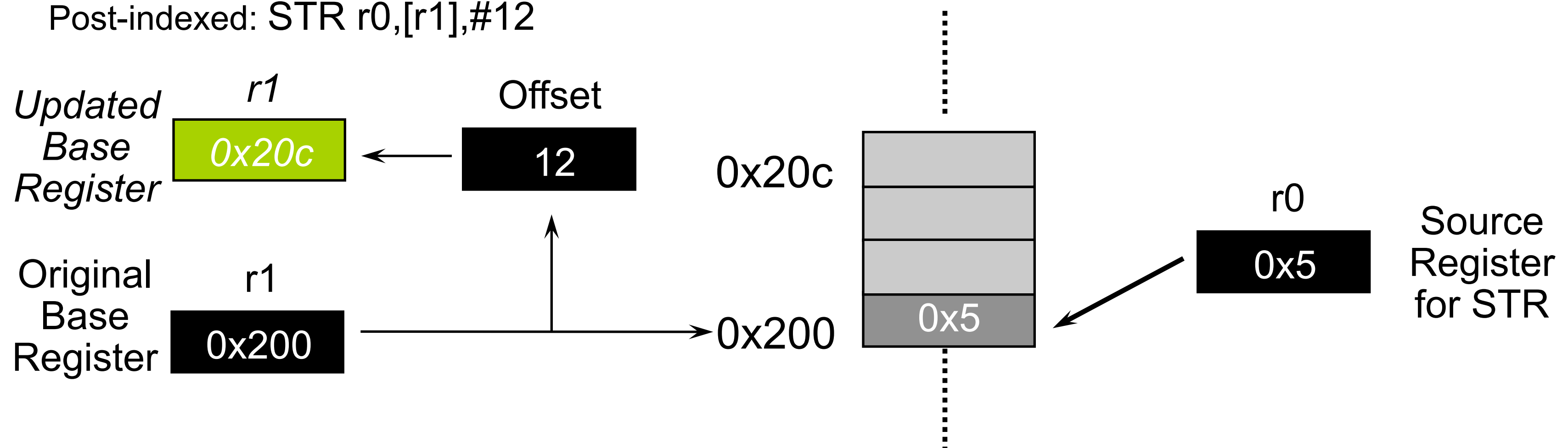
Pre or Post Indexed Addressing?

- Pre-indexed: `STR r0,[r1,#12]`



Auto-update form: STR r0,[r1,#12]!

- Post-indexed: STR r0,[r1],#12



LDM / STM operation

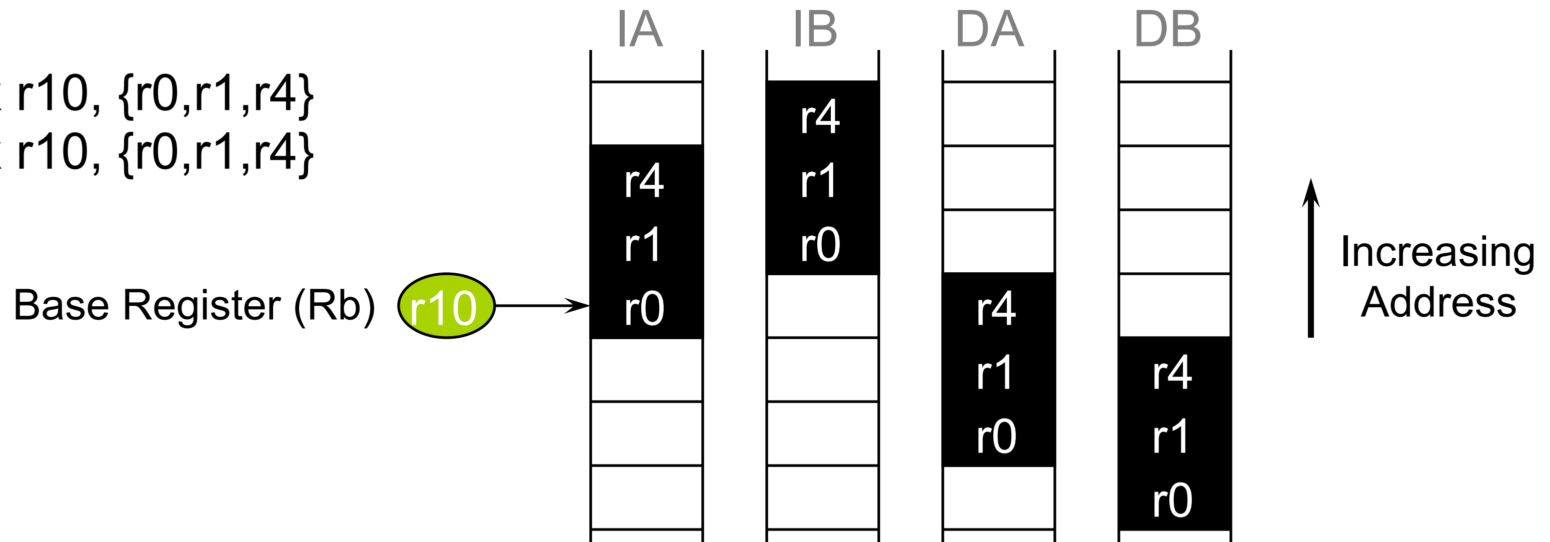
- Syntax:

  <LDM|STM>{<cond>}<addressing_mode> Rb{!}, <register list>

- 4 addressing modes:

LDMIA / STMIA	increment after
LDMIB / STMIB	increment before
LDMDA / STMDA	decrement after
LDMDB / STMDB	decrement before

LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}



Multiple Register Load and Store Instructions

- Used for transferring large quantities of data
- Usage: procedure entry & exit
- LDMIA r1, {r0, r2, r5}; r0 := mem_{32}[r1]

  ;r2 := mem_{32}[r1+4]

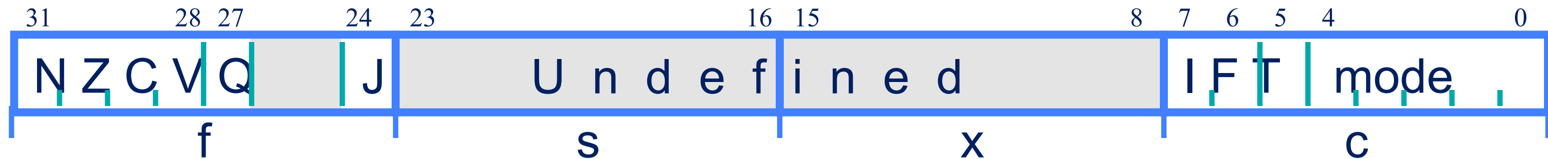
  ; r5 := mem_{32}[r1+8]

r1 should be aligned

If you put r15 in {}, it will change control flow

You can combine with **! also.**

PSR Transfer Instructions



- MRS and MSR allow contents of CPSR / SPSR to be transferred to / from a general purpose register.
- Syntax:
  - **MRS{<cond>} Rd,<psr>   ; Rd = <psr>**
  - **MSR{<cond>} <psr[_fields]>,Rm ; <psr[_fields]> = Rm**

  where
  - <psr> = CPSR or SPSR
  - [_fields] = any combination of 'fsxc'
- Also an immediate form
  - **MSR{<cond>} <psr_fields>,#Immediate**
- In User Mode, all bits can be read but only the condition flags (_f) can be written.

Exceptions/Interrupts

- 1. Exceptions generated as the direct effect of executing an instruction
  - Software interrupt, undefined instructions, prefetch aborts
- 2. Exceptions generated as a side-effect of an instruction
  - Memory fault during a load or store data access
  - Unaligned access
- 3. Exceptions generated externally, unrelated to the instruction flow. Reset, IRQ, and FIQ

Exception/Interrupt

- Interrupt:
  - It handles as soon as the current instruction is finished
  - E.g.) External events, Fast interrupt (FIQ)
- Exception
  - It handles immediately
    - E.g.) page faults, unaligned accesses, undefined opcode

Sequences of Actions

- 1) changes to the operation mode corresponding to the particular exception
- 2) saves the next PC address into the corresponding r14 register.
- 3) Saves the old value of CPSR in the SPSR of the new mode
- 4) Disables IRQs by setting bit 7 of the CPSR
  - For a fast interrupt, disables further fast interrupt by setting bit 6 of the CPSR. (no nested fast interrupts!)
- 5) Set PC address to the corresponding interrupt vector table

Exception Vector Address

Exception	Mode	Vector Address	Priority
Reset	SVC	0x00000000	1
Undefined instruction	UND	0x00000004	6
Software interrupt (SWI)	SVC	0x00000008	6
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C	5
Data abort (data access memory fault)	Abort	0x00000010	2
IRQ (normal interrupt)	IRQ	0x00000018	4
FIQ (Fast interrupt)	FIQ	0x0000001C	3

- Vector address contains a branch to the relevant routine, except FIQ
  - No space to put code.
- FIQ code can start immediately because it has the highest vector address.

Registers

- Two banked registers to hold the return address and a stack pointer
- Stacks are used to store registers
  - Callee based register saving
- FIQ→ additional registers
  - Why? To save time to save registers

Exception Return

- After the exception handler, the hardware just starts from the user mode.
- Software must
  - – Restore the modified registers
  - – CPSR must be restored from the appropriate SPSR
  - – PC must be changed back to the relevant instruction address in the user instruction stream
    - These two cannot happen independently

Atomic Operations

- Return using a link register (r14)

  MOVS pc, r14

  Opcode+S → PC is the destination register

  S bit is set, a branch occurs and the SPSR of the current mode is copied to the CPSR

- Return using a stack

  LDFMD sp!, {r0-r12, pc}^

  – The ^ qualifier specifies that the CPSR is restored from the SPSR. It must be used only from a privileged mode.

Exercise