- **임베디드 시스템**

마이크로 프로세서 혹은 마이크로 컨트롤러를 내장(embedded)하여 개발자가 지정한 기능만을 수행하는 장치. 보다 큰 시스템의 일부이거나 독립된 시스템으로서 특정 업무를 수행하기 위한 HW와 SW를 포함하고 있는 응용 시스템.

- **임베디드 시스템 응용 분야별 특징**

	실시간성	입출력	GUI	CPU 처리	저 소비 전력 요구
제어	**H**	多	x	M	M
가전	L	少	대형	**H**	L
단말	L	少	중소형	M	**H**
통신장비	M	**多**	x	M	M

- **실시간 시스템 (Real-Time System)**

특정 반응에 대해 정해진 시간 내에 행동 할 수 없을 때 문제가 발생하는 시스템. 결과 산출 시간에도 적시성(timeliness)을 가지며 외부 자극에도 예측 가능한(predictable) 방식으로 반응

적시성이란, 열악한 환경하에서도 반응에 요구되는 한계 시간(deadline)이 내에 논리적으로도 정확한 출력 값을 산출해 내는 것이다.

➢ Hard RTS

제어작업이 deadline을 어기는 경우 시스템에 심각한 영향(failure)을 주는 time-critical 속성을 지닌 시스템. (Ex. 항공기, 우주 왕복선, 자동차 ...)

➢ Soft RTS

Deadline을 어긴 단위 제어 작업의 무효화로 시스템의 평균적 성능에 미세한 영향을 주는 시스템. (Ex. 컴퓨터, 정보기기, 네트워크 기기 ...)

- **마이크로 프로세서의 특징**

동일한 logic을 활용하여 다양한 기능을 구현. / 동일 계열 제품(product)들의 design을 단순화. / 기능을 구동시키는데 비교적 많은 logic 사용. / 제품의 처리 속도를 빠르게 하는 데 유용. / 소비 전력을 효율적으로 제어하는 것이 중요. / Never terminate.

- **임베디드 시스템**

특정 목적으로 구성된 HW 위에 SW를 내장하여 최적화 시킨 시스템.

- **임베디드 SW**

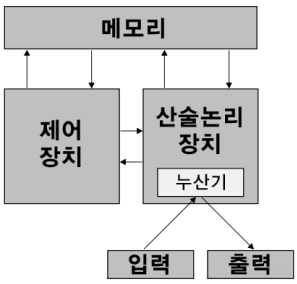
임베디드 시스템에 탑재되는 시스템 SW, 미들웨어, 응용 SW를 총칭

운영체제(OS) / 미들웨어 / 응용 SW / 시스템 SW / SW 개발 프레임워크

- **임베디드 시스템 설계(Design)시 고려사항**

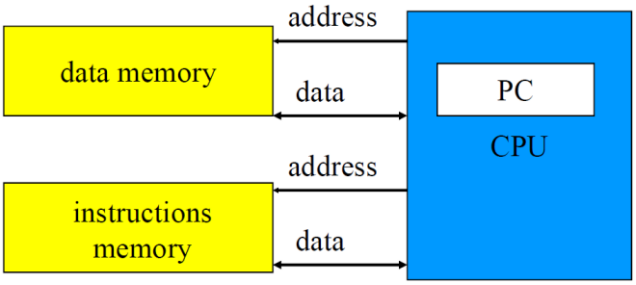
성능(Performance) / 기능(Functionality) 및 UI / 제작 비용 / 소비 전력의 효율성 / 크기

- **Von Neumann Architecture**



일반 data와 명령어가 동일한 bus를 공유하는 형태. CPU가 메모리에서 명령어를 가져온다. CPU는 PC와 IR 그리고 범용 레지스터 등을 가진다.

- **Harvard Architecture**



Data용 버스와 명령어용 버스가 분리되어 있는 형태. 명령어 처리를 끝내자마자 다음 명령어를 fetch할 수 있음.

Most DSP devices user Harvard architecture for streaming data.

- **Pipelining**

동일 시간대에 여러 개의 다른 instruction이 병렬적으로 수행됨. 많은 instruction을 처리하는 경우 유리함.

- **Feature of Pipelining**

$$\text{Potential Speedup} = \text{Number of Pipeline Stages}$$

Unbalanced lengths of pipe stages reduce speedup.

- **5 Stages of Instruction Execution**

1. **Fetch**: Fetch instruction.
2. **Decode**: Decode the instruction.
3. **Execute**: Execute the operation.
4. **Memory**: Access an operand in data memory
5. **Write**: Write the result into a register.

- **Pipeline Performance**

Clock cycle is determined by the time required for the slowest pipe stage.

With perfectly balanced pipeline stages,

$$\text{Time Between instructions}_{\text{pipelined}} = \frac{\text{Time Between instructions}_{\text{non-pipelined}}}{\text{Number of pipe stages}}$$

Speedup of k-stage pipeline with clock cycle time t for n instructions,

$$\text{Speedup} = \frac{n \cdot k \cdot t}{(k-1)t + n \cdot t} = \frac{n \cdot k \cdot t}{k \cdot t + (n-1)t} \overset{n \to \infty}{=} k$$

- **Complex Instruction Set Computer (CISC)**

전통적인 명령어 처리방식. 많은 수의 명령어와 주소 모드. 따라서 instruction decoding time이 큼. 명령어 처리가 HW의존적이므로 전력 소모가 큼. Register의 수가 적으므로 메모리 접근 빈도가 높음.

- **Reduced Instruction Set Computer (RISC)**

명령어의 개수를 줄임으로써 HW구조를 보다 간단하게 설계. 고정 길이 명령어를 사용하여 빠른 해석. 많은 수의 register를 사용하여 메모리 접근 수를 줄임.

메모리 접근은 `load`, `store`등 한정된 명령어로 제한하여 회로를 단순화하고 불필요한 메모리 접근을 방지함. 따라서 SW의존적이므로 전력 소모가 적어 대부분의 임베디드 프로세스에 적합하다.

- **CISC vs. RISC Ex. `X=(A+B)*(C+D)`**

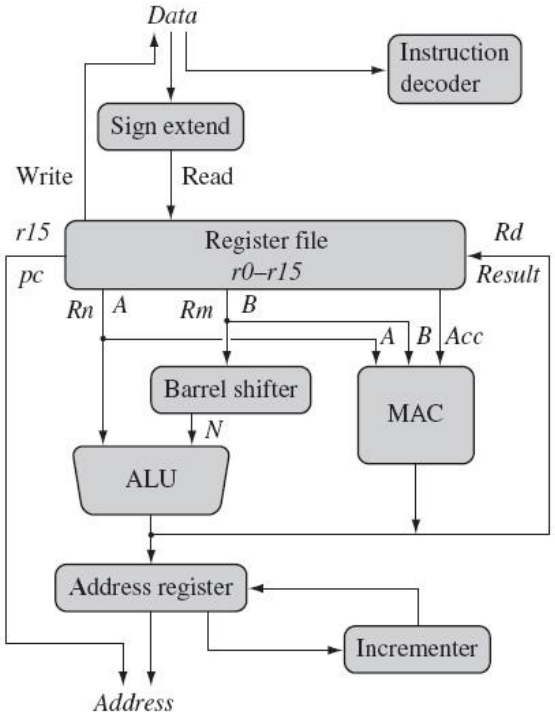
➢ In CISC machine

ADD	R1	A	B
ADD	R2	C	D
MUL	X	R1	R2

➢ In RISC machine

LOAD	R1	A	
LOAD	R2	B	
LOAD	R3	C	
LOAD	R4	D	
ADD	R1	R1	R2
ADD	R3	R3	R4
MUL	R1	R1	R3
STORE	X	R1	

- **ARM Core Data Flow Model**



32bit 레지스터 / Sign extend / Register file: 16개의 User mode register / Rn, Rm: operand 저장 reg. / Barrel Shifter / Multiple Accumulate Unit(MAC, 누산기)

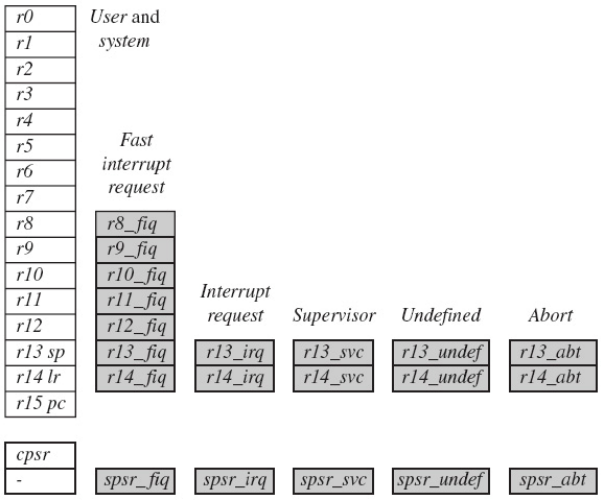
- **ARM Data Types**

Byte(8) / Halfword(16) / Word(32) / Doubleword(64)

- **ARM Instruction Sets**

  ARM(32) / Thumb(16) / Thumb-2(16, 32) / Jazelle(8)

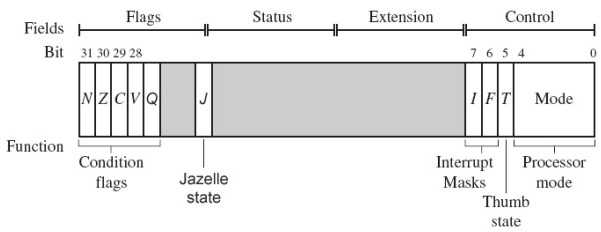
- **ARM Register Set**



  ➢ ARM has 37 registers which are 32bit long

  - 1 for Program Counter(PC)
  - 1 for Current Program Status Reg.(CPSR)
  - 5 for Saved Program Status Reg.(SPSR)
  - 30 for General purpose Reg.
  - 20 for Banked Reg. (gray of above figure.)

- **Program Status Register(PSR)**



  ➢ CPSR

  코어의 내부 동작을 모니터링하고 제어. 특권 모드일 때에만 read/write 가능. 일반 모드에서는 condition flag만 read/write 가능하고 control flag는 read만 가능.

  ➢ SPSR

  프로세서의 동작 모드가 변경될 때 활성화됨. 현재의 CPSR 내용을 저장.

- **ARM Processor Mode**

  ➢ `usr` (User)

  Normal program execution mode.

  ➢ `irq` (Interrupt Request)

  일반적인 I/O 장치로부터 입력이 들어오면 반응하는 regular interrupt 모드.

  ➢ `fiq` (Fast Interrupt Request)

  `irq`와 유사하나 빠른 데이터 전송 및 처리를 할 수 있도록 하는 interrupt 모드.

  ➢ `svc` (Supervisor Calls)

  프로세서가 reset될 때 진입, 운영체제의 커널이 동작.

  ➢ `abt` (Abort)

  메모리 access 실패 시 진입.

  - `und` (Undefined)

    정의되지 않거나 지원하지 않는 명령어를 만날 때 진입.

  - `sys` (System)

    특권을 갖는 user 모드로, CPSR에 읽기 및 쓰기가 가능. SW에 의해 요청.

- **Condition Flags**

  ALU의 연산 결과를 기록

  ➢ **N**: Negative result from ALU

  ➢ **Z**: Zero result from ALU

  ➢ **C**: ALU operation carried out

  ➢ **V**: ALU operation overflowed

  ➢ **Q**: Overflow & Saturation

  다른 것과 달리 `Q` bit가 일단 `1`로 set 되면 바뀌지 않음. `MRS, MSR`등의 명령어를 통해 `PSR`을 다시 setting해야함.

- **Pipeline Hazards**

  ➢ Structural Hazard

  A Hazard due to a resource conflict. Some functional unit is not fully pipelined and resources do not support pipeline environments.

  Ex. A machine that has a shared memory for data and instructions.

Clock cycle number							
1	2	3	4	5	6	7	8
IF	ID	EX	**MEM**	WB			
	IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB

  ➢ Data Hazard

  RAW / WAR / WAW

  ➢ Control Hazard

  Control hazards arise from the pipelining of branches and other instructions that change the PC

Clock cycle number							
1	2	3	4	5	6	7	8
IF	ID	EX	**MEM**	WB			
	IF	**-**	**-**	IF	ID	EX	MEM
					IF	ID	EX
						IF	ID

- **ARM Exception/Interrupt Process**

  i. CPSR reg. 내용이 SPSR_<mode> reg.로 복사.
  ii. CPSR bit를 적절하게 설정.
    - Thumb state에서 동작 중이었으면 ARM state로 변경
    - 해당 exception에 맞게 프로세서 mode bit가 변경됨.
    - Disable interrupt masks.
  iii. Exception 종류 후 복귀 주소값(현재 PC의 값)은 LR_<mode> reg.에 저장.
  iv. PC에 특정 메모리 주소값(Vector Table)이 입력됨
  v. 현재 instruction이 종료된 후 PC에 저장된 주소값 위치로 이동
  vi. Exception/Interrupt를 처리
  vii. Restore CPSR from SPSR_<mode>
  viii. Restore PC from LR_<mode>

- **ARM Exception Vector table**

Exception	Shorthand	Vector addr.
Reset	RESET	0x0000'0000
Undefined instruction	UNDEF	0x0000'0004
Software interrupt	SWI	0x0000'0008
Prefetch abort	PABT	0x0000'000c
Data abort	DABT	0x0000'00010
(Reserved)	-	0x0000'0014
Interrupt request	IRQ	0x0000'0018
Fast interrupt request	FIQ	0x0000'001c

- **ARM Exceptions**

  ➢ Reset

  전원 공급 시 처음 실행되는 명령어. Initialization code로 분기

  ➢ Undefined instruction

  When the processor cannot decode an instruction.

  ➢ Software interrupt

  User 모드에서 동작 중인 프로그램이 발생시킴. Ex. 프로그램이 OS에 특정 서비스를 요청

  ➢ Prefetch abort

  When the processor attempts to fetch an instruction from an address without the correct access permission.

  ➢ Data abort

  When an instruction attempts to access data memory without the correct access permission.

  ➢ Interrupt request (IRQ)

  When an external hardware interrupts the normal execution flow of the processor.

  ➢ Fast interrupt request (FIQ)

  When a hardware requiring faster response times interrupts the normal execution flow of the processor. 신속 처리→Vector table 가장 아래쪽.

- **ARM Exception Priority**

Exception	Priority	I bit	F bit
Reset	1	1	1
Data abort	2	1	
Fast interrupt request	3	1	1
interrupt request	4	1	
Prefetch abort	5	1	
Software interrupt, Undef	6	1	

- **Cache**

  전반적인 시스템 성능은 향상시키나 예측성, 즉 데이터나 명령어를 읽고 쓰는데 걸리는 시간을 예측할 수 있는 능력이 떨어진다.

- **TCM**

  예측 성을 향상시키키기 위한 메모리다. 명령어나 데이터를 읽는데 데 필요한 클럭 사이클을 보장한다. 코어 가까이에 위치한 빠른 속도의 SRAM.

- **Non-protected memory**

  일반 메모리 관리 장치. Application으로부터의 보호를 필요로 하지 않는 작고 간단한

- **Memory Management Unit (MMU)**

  완전한 보호를 제공하는 메모리 관리 장치. ARM에서 사용되는 가장 복잡한 메모리 관리 장치. Multitasking을 지원하는 복잡함 platform OS 필요.

- **Coprocessors**

  ARM 명령어 세트에 없는, 확장된 명령어를 처리하는 프로세서. Decode 단계에서 extended instruction set임이 밝혀지면 coprocessor로 전달됨

  Coprocessor가 존재하지 않거나 해독 불가능한 instruction이면 ARM은 Undefined Exception을 발생시킴.

- **ARM Data Processing Instructions: Move**

  `MOV{<cond>}{S} Rd, N`

  Copies `N` into a destination reg. `Rd / N` is a reg. or immediate value.

```
MOV r7, r5 // r7 == r5
MVN r7, r5 // r7 == ~r5
MOVS r7, ,5 //
```

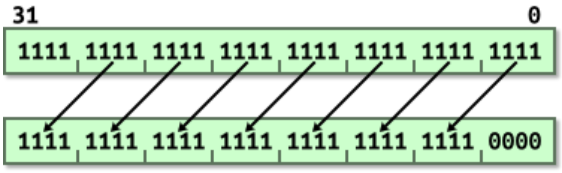
- **ARM Data Processing Instructions: Barrel Shifter**

  `x <operator> y`

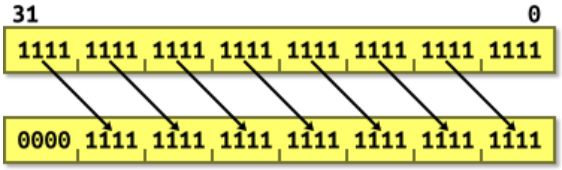
  x is shifted by the amount of y

  ➢ `operator` for...

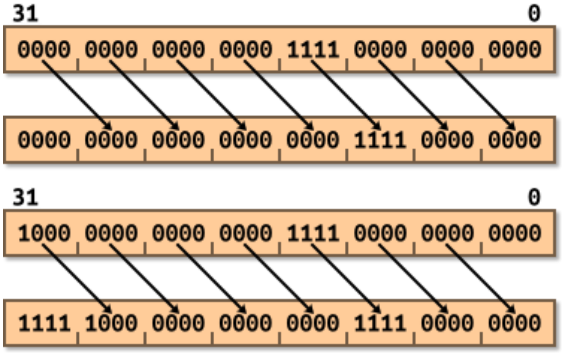
  - LSL: logical shift left



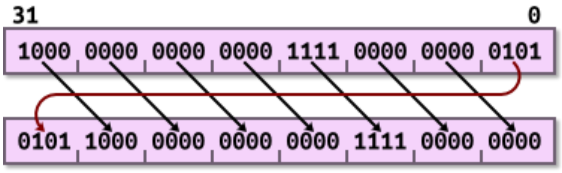
  - LSR: logical shift right



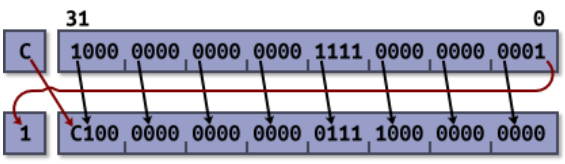
  - ASR: arithmetic right shift



  - ROR: rotate right



  - RRX: rotate right extended




```
MOV r7, r5 LSL #2 // r7 == r5 * 4
```

- **ARM Data Processing Instructions: Arithmetic**

```
<instruction>{<cond>}{S} Rd, Rn, N
```

32bit signed, unsigned값의 덧셈, 뺄셈

➢ instruction for...

- `ADD: Rd = Rn + N`
- `ADC: Rd = Rn + N + carry`
- `SUB: Rd = Rn - N`
- `SBC: Rd = Rn – N – ~carry`
- `PSB: Rd = N - Rn`
- `PSC: Rd = N – Rn - ~carry`

```
ADD r0, r1, r1, LSL #1 // r0 == r1 * 3
```

- **ARM Data Processing Instructions: Logical**

<instruction>{<cond>}{S} Rd, Rn, N

➢ instruction for...

- `AND: Rd = Rn & N:`
- `ORR: Rd = Rn | N`
- `EOR: Rd = Rn ^ N // exclusive OR`
- `BIC: Rd = Rn & ~N // bit clear`

```
r0 = 0b0000
r1 = 0b1111
r2 = 0b0101
BIC r0, r1, r2 // r0 == 0b1010
```

- **ARM Data Processing Instructions: Comparison**

<instruction>{<cond>}{S} Rn, N

reg.에 저장된 값의 비교

➢ instruction for...

- `CMP: Rn – N // compare positive`
- `CMN: Rn + N // compare negative`
- `TEQ: Rn ^ N // test for equality of two 32-bit values`
- `TST: Rn & N // test bits of a 32-bit value`

```
cpsr = nzcvqiFt_USER
r0 = 4
r9 = 4
CMP r0, r9 // cpsr == nZcvquiFt_USER
```

- **ARM Data Processing Instructions: Multiply**

```
MLA{<cond>}{S} Rd, Rm, Rs, Rn
MUL{<cond>}{S} Rd, Rm, Rs
```

- `MLA: Rd = (Rm * Rs) + Rn`
  Multiply and accumulate
- `MUL: Rd = Rm * Rs`

- **ARM Branch Instructions**

```
<B | BL>{<cond>} label
BX{<cond>} label
BLS{<cond>} label | Rm
```

실행의 흐름을 변경하거나 특정 루틴을 호출

- `B: branch`
  `pc = label`
  복귀하지 않고 label부터 수행. Sub procedure call에 사용.

- `BL: branch with link`
  `pc = label`
  `lr = address of the next instruction after the BL`
  label 수행 후 복귀. Sub function call에 사용.

```
BL          subroutine
CMP         r1, #5 // compare r1 with 5
MOVEQ       r1, #0 // if(r1 == 5) r1 = 0
subroutine
MOV         pc, lr // return
```

- **ARM Load-Store Instruction: Single-Reg. Transfer**

```
<op>{<cond>}{<type>} Rd, [Rn {, #offset}]
```

➢ op for...

- `LDR: Rd = mem32[address]`
- `STR: Rd = mem32[address]`

➢ type for...

- `B: unsigned byte`
- `SB: signed byte // LDR only`
- `H: unsigned half-word`
- `SH: signed half-word // LDR only`

- **ARM Load-Store Instruction: Swap**

`SWP{B}{<cond>} Rd, Rm, [Rn]`

Exchange a memory content and a register content.

{B} is for byte.

```
tmp = mem32[Rn]
mem32[Rn] = Rm
Rd = tmp
```

SWAP cannot be interrupted by any other instructions or any other bus access.

Pre-condition

```
mem32[0x90’00] = 0x12’34’56’78
r0 = 0x00’00’00’00
r1 = 0x11’11’22’22
r2 = 0x00’00’90’00

SWP r0, r1, [r2]
```

Post-condition

```
mem32[0x90’00] = 0x11’11’22’22
r0 = 0x12’34’56’78
r1 = 0x11’11’22’22
r2 = 0x00’00’90’00
```

- **ARM Software Interrupt Instruction**

SWI{<cond>} SWI_number

Causes a software interrupt exception. Provides a mechanism for applications to call OS routines.

```
lr_svc = addr. of instr. following the SWI
spsr_svc = cpsr
pc = vectors + 0x8
cpsr mode = SVC
cpsr I = 1 // mask IRQ interrupts
```

Pre-condition

```
cpsr = nzcVqift_USER
pc = 0x00’00’80’00
lr = 0x00’00’80’00

0x80’00     SWI 0x12’34’56
```

Post-condition

```
cpsr = nzcVqIft_SVC
spsr = nzcVquft_USER
pc = 0x00’00’00’08
lr = 0x00’00’80’04
```

- **ARM Program Status Register Instructions**

  - `MRS{<cond>} Rd, <cpsr|spsr>`
    Move psr to Rd.
  - `MSR{<cond>}<cpsr|spsr>_<fields>, Rm`
    Move Rm value to psr.
  - `MSR{<cond>}<cpsr|spsr>_<fields>, #immd`
    Move an immediate value to psr.
  - `fileds for…`
    - `c`: control
    - `x`: extension
    - `s`: status
    - `f`: flags

Pre-condition

```
cpsr = nzcvqIFt_SVC

MRS r1, cpsr
BIC r1, r1, #0x80 // 0b10’00’00’00
MSR cpsr_c, r1
```

Post-condition

```
cpsr = nzcvqiFt_SVC
```

- **Load Address Pre-indexing**

Pre-condition

```
r0 = 0x0000’0000
r1 = 0x0000’9000
mem32[0x0000’9000] = 0x0101’0101
mem32[0x0000’9004] = 0x0202’0202

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

Post-condition

```
r0 = 0x0202’0202
r1 = 0x0000’9000
```

- **Load Address Post-indexing**

Pre-condition

```
r0 = 0x0000’0000
r1 = 0x0000’9000
mem32[0x0000’9000] = 0x0101’0101
mem32[0x0000’9004] = 0x0202’0202
```

LDR r0, [r1], #4

Post-condition

```
r0 = 0x0101’0101
r1 = 0x0000’9004
```

- **Load Address Per-indexing with Write-back**

  Pre-condition

```
r0 = 0x0000’0000
r1 = 0x0000’9000
mem32[0x0000’9000] = 0x0101’0101
mem32[0x0000’9004] = 0x0202’0202

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

  Post-condition

```
r0 = 0x0202’0202
r1 = 0x0000’9004
```

- Writing Loops Efficiently

  Use loops that count down to zero.

```
int checksum_v5(int *data){
  unsigned int i;
  int sum = 0;

  for(i = 0; i < 64; i++)
    sum += *(data++);

  return sum;
}
```

```
checksum_v5
  MOV r2, r0           ; r2 = data
  MOV r0, #0           ; sum = 0
  MOV r1, #0           ; i = 0
checksum_v5_loop
  LDR r3, [r2],#4      ; r3=*(data++)
  ADD r1, r1, #1       ; i++
  CMP r1, #0x40        ; compare i, 64
  ADD r0, r3, r0       ; sum += r3
  BCC checksum_v5_loop; if (i<64) loop
  MOV pc, r14          ; return sum
```

```
int checksum_v6(int *data) {
  unsigned int i;
  int sum = 0;

  for (i=64; i!=0; i--)
    sum += *(data++);

  return sum;
}
```

```
checksum_v6
  MOV r2, r0           ; r2 = data
  MOV r0, #0           ; sum = 0
  MOV r1, #0x40        ; i = 64
checksum_v6_loop
  LDR r3, [r2], #4     ; r3=*(data++)
  SUBS r1, r1, #1      ; i--, set flags
  ADD r0, r3, r0       ; sum += r3
  BNE checksum_v6_loop; if (i != 0) loop
  MOV pc, r14          ; return sum
```

  Use do-while loops rather than for loops if there is least one iteration.

```
int checksum_v7(int *data,
                unsigned int N){
  int sum = 0;
  for (; N != 0; N--)
    sum += *(data++);

  return sum;
}
```

```
checksum_v7
  MOV r2, #0           ; sum = 0
  CMP r1, #0           ; compare N, 0
  BEQ checksum_v7_end ; if(N==0) goto end
checksum_v7_loop
  LDR r3, [r0],#4      ; r3=*(data++)
  SUBS r1, r1, #1      ; N-- and set flags
  ADD r2, r3, r2       ; sum += r3
  BNE checksum_v7_loop; if (N!=0) loop
Checksum_v7_end
  MOV r0, r2           ; r0 = sum
  MOV pc, r14          ; return r0
```

```
int checksum_v8(int *data,
                unsigned int N){
  int sum = 0;
  do {
    sum += *(data++);
  } while (--N != 0);

  return sum;
}
```

```
checksum_v8
  MOV r2, #0           ; sum = 0
checksum_v8_loop
  LDR r3, [r0],#4      ; r3=*(data++)
  SUBS r1, r1, #1      ; N-- and set flags
  ADD r2, r3, r2       ; sum += r3
  BNE checksum_v8_loop; if (N!=0) loop
  MOV r0, r2           ; r0 = sum
  MOV pc, r14          ; return r0
```

-

-