SEP

LEZIONE 1 DURANTE IL CORSO USEREMO IL MICROCONTROLLORE ATMEL ATMEGA 32

- I PROCESSORI ATMEL HANNO IL GRANDE VANTAGGIO CHE SONO DEI PROCESSORI NORMALI. AD ESEMPIO IL PIC A 8 bit E' STRANO", CIOE' E' MENO GENERALE E QUINDI E' DIFFICILE TRASPORTARLO SU ALTRI PROCESSORI. MENTRE I PROCESSORI ATMEL HANNO CARATTERISTICHE SIMILI ALLA MAGGIOR PARTE DEI PROCESSORI
- NON E' IL µCONTROLLORE PIÙ USATO, QUELLO PIÙ USATO E' ARM (PRESENTE ANCHE NEGLI SMARTPHONE). MA QUESTO E' TROPPO DIFFICILE PER I PRINCIPIANTI

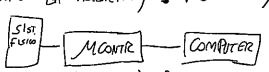
⇒ AVR E' COMPATTO, CIOE' TUTTE LE INFORMAZIONI NECESSARIE PER USARLO SI RICORDANO A MENTE, MENTRE PER USARE ARM SI DEVE FAR USO DI UN MANUALE DA PIÙ DI 1000 PAGINE

DURANTE LE ESERCITAZIONI USEREMO LA SCHEDA STK600 (80€) CHE HA IL PROCESSORE E UN PO' DI ELETTRONICA SOPRA

NOI CI CONCENTREREMO SU APPLICAZIONI CHE RICHIEDONO LA SCRITTURA DI FIRMWARE. UN FIRMWARE E' PROGRAMMA CHE SI SCRIVE DENTRO IL PROCESSORE CHE RESTA LÌ (DI SOLITO NON SI CAMBIA DURANTE LA VITA DI UN PRODOTTO)

APPLICAZIONI → AUTOMAZIONE GENERALE

- ESEMPI DI SISTEMI OPERATIVI REAL-TIME (NON HANNO LA PRIORITÀ) : FREERTOS, CHIBIOS. MA IL PROBLEMA E' CHE AUMENTA IL COSTO DI SVILUPPO POICHÉ SONO COMPLESSI DA USARE

⇒ ALLORA SPESSO SI DIVIDE L'APPLICAZIONE 

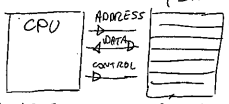
E IL FIRMWARE E' NEL µCONTROLLORE INVECE DI FARE UN SUPER-FIRMWARE CHE RICHIEDE UN SISTEMA OPERATIVO REAL-TIME (DIFFICILE) ⇒ SI FA UN PROGRAMMA NORMALE (FACILE), NON REAL-TIME, SUL COMPUTER E UN FIRMWARE NEL µCONTROLLORE, PRIVO DI SISTEMA OPERATIVO ⇒ REAL-TIME

- µCONTROLLORI SUL MERCATO:
  - VECCHIA GENERAZIONE { 8051, 65H11, Z80 (PROCESSORE VERO E PROPRIO) } → SVANTAGGIO: SI PUÒ PROGRAMMARE SOLO IN ASSEMBLER PERCHÉ HA POCHI REGISTRI
  - PIC (NON VECCHIO) → SVANTAGGIO
  - NUOVA GENERAZIONE { ATMEL AVR (PROGETTATO PER ESSERE PROGRAMMATO IN C), TEXASI MSP430, ARM (NECCHIA GENERAZIONE ('82) MA NATO COME PROCESSORE DA PC ⇒ TANTI REGISTRI) }

MASTER CO
Tel. 050 8312126
Cell. 388 9837745

ATMEL ATMEGA 32

IL CALCOLATORE NASCE PIÙ O MENO DAGLI STUDI DI VON NEUMANN (~'46) CHE DISSE CHE ERA COMODO CHE UN CALCOLATORE POTESSE CONSERVARE TUTTO (SIA I DATI CHE PROGRAMMI) IN UNA SOLA MEMORIA (RAM)



NELLA RAM CI SONO TANTE CELLE, OGNUNA CON UN SUO INDIRIZZO, CHE CONTENGONO UN NUMERO BINARIO A 8, 16 bit. C'E' UN SISTEMA DI CODIFICAZIONE PER CUI TUTTI I DATI E TUTTE LE ISTRUZIONI SONO TRASFORMATE IN NUMERI CHE VENGONO MESSE NELLE CELLE (O COME ISTRUZIONE O COME DATO)

QUALCHE ANNO DOPO, QUALCUNO HA DETTO CHE QUESTA STRUTTURA E' LIMITANTE POICHÉ SI PUÒ LEGGERE E SCRIVERE UNA CELLA ALLA VOLTA

⇒ PER CUI VIENE FUORI L'ARCHITETTURA DI HARVARD, IN CUI SONO PRESENTI 2 (O PIÙ) MEMORIE. SVILUPPATA PER I DSP (DIGITAL SIGNAL PROCESSOR) CON 2 MEMORIE (UNA PER LE ISTRUZIONI E UNA PER I DATI) POSSO PRENDERE ISTRUZIONI E DATI SIMULTANEAMENTE

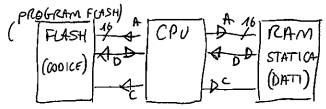
⇒ I PROGETTISTI DI ATMEL HANNO SCELTO L'ARCHITETTURA HARVARD, NON PERCHÉ SONO DSP (NON HANNO BISOGNO DI LEGGERE DATI A VELOCITÀ ELEVATE, TRA L'ALTRO LEGGONO UN DATO ALLA VOLTA), MA PERCHÉ POSSONO COSTRUIRE LE DUE MEMORIE CON TECNOLOGIE DIVERSE

⇒ RAM (PER I DATI) E FLASH (PER LE ISTRUZIONI)

LE MEMORIE HANNO 3 FUNZIONI: LETTURA, SCRITTURA, CONSERVAZIONE. PER LEGGERE E SCRIVERE RICHIEDONO NECESSARIAMENTE DELL'ENERGIA, MENTRE LA CONSERVAZIONE NON E' DETTO

- MEMORIE
  - VOLATILI (RICHIEDONO ENERGIA ANCHE PER CONSERVARE DATI)
    - STATICHE (FINCHÉ DURA L'ENERGIA, IL DATO RESTA. LE PIÙ COMODE)
    - DINAMICHE (ANCHE IN PRESENZA DI ENERGIA NON SONO IN GRADO DI MEMORIZZARE DATI A LUNGO TERMINE. TIPICAMENTE I DATI RIMANGONO PER 4ms, QUINDI SI DEVE FARE UN REFRESH CONTINUAMENTE)
  - NON VOLATILI
    - ROM (SI PUÒ SOLO LEGGERE)
    - PROM (SI PUÒ SCRIVERE UNA VOLTA SOLA)
    - EPROM (SI PUÒ CANCELLARE ESPONENDOLE AGLI UV)
    - EEPROM (CANCELLABILI ELETTRICAMENTE) → PRESENTE NELL'ATMEL
    - FLASH (CANCELLAZIONE DI TUTTE LE CELLE)

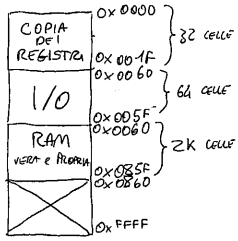
ATMEL



- ALL'INTERNO DELLA CPU ABBIAMO L'UNITÀ LOGICA ARITMETICA (ALU) CHE E' QUELLA CHE FA I CONTI SIA LOGICI CHE ARITMETICI; POI ABBIAMO IL DECODIFICATORE DI ISTRUZIONE CHE PRENDE LE ISTRUZIONI DALLA MEMORIA FLASH E LE ESEGUE E QUINDI COORDINA GLI ALTRI ORGANI; POI ABBIAMO I REGISTRI CHE SONO COME UNA PICCOLA MEMORIA
- LA MEMORIA DEI DATI E' UNA MEMORIA A 8 bit, QUINDI OGNI CELLA CONTIENE UN NUMERO A 8 bit { RAPPRESENTATI CON INTERI SENZA SEGNO ⇒ 0 ÷ 255; RAPPRESENTATI CON INTERI IN COMPLEMENTO A 2 ⇒ −128 ÷ 127 }

GLI INDIRIZZI SONO A 16 bit, QUINDI E' POSSIBILE INDIRIZZARE FINO A $2^{16} = 65536$ CELLE. MA NON TUTTE LE CELLE SONO FISICAMENTE IMPLEMENTATE, MA SOLO 2K = 2048 SONO RAPPRESENTATE

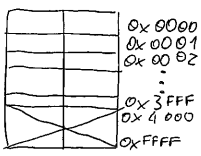
MAPPA DI MEMORIA



CELLE NON IMPLEMENTATE

- SE UNO SCRIVE IN UN INDIRIZZO NON IMPLEMENTATO, IL DATO VA PERDUTO. SE UNO LEGGE UN INDIRIZZO NON IMPLEMENTATO, SI LEGGE UN DATO FITTIZIO DI TUTTI 1 ⇒ FF = 255
- LE CELLE DI MEMORIE IN I/O CONTROLLANO L'HARDWARE, CIOE' SONO COLLEGATE DIRETTAMENTE COL MONDO ESTERNO (PERIFERICHE)

MAPPA DELLA FLASH



- ANCHE GLI INDIRIZZI DELLA FLASH SONO A 16 bit, MA LE CELLE SONO A 16 bit ⇒ IN QUESTO PROCESSORE LE ISTRUZIONI SONO A 16 bit
- NON E' POSSIBILE ACCEDERE A MEZZA CELLA (8 bit) E L'INDIRIZZO SI RIFERISCE A TUTTA LA CELLA, CIOE' A TUTTI I 16 bit. QUESTA COSA NON E' STANDARD, LA MAGGIOR PARTE DEI PROCESSORI HANNO L'INDIRIZZO 0 PER I PRIMI 8 bit DELLA PRIMA CELLA DELLA FLASH, E L'INDIRIZZO 1 PER I SECONDI 8. IN QUESTO CASO, SE NON FOSSE POSSIBILE ACCEDERE A MEZZA CELLA, VORREBBE DIRE CHE L'USO DEGLI INDIRIZZI DISPARI SAREBBERO INUTILI. INVECE L'ATMEL ATMEGA 32 USA UN INDIRIZZO PER TUTTI I 16 bit

⇒ WORD ADDRESS

L'ATMEGA 32, IL 32 STA PER 32K byte IMPLEMENTATI (CHE SONO 16K WORD)

MASTER COPY
Tel. 050 8312126
Cell. 388 9837745

SEP

- DENTRO IL PROCESSORE C'È UN REGISTRO PARTICOLARE, A 16 bit, CHIAMATO PROGRAM COUNTER PC [|] E TIENE CONTO A QUALE ISTRUZIONE SIAMO ARRIVATI, CIOÈ DOPO L'ESECUZIONE DI CIASCUNA ISTRUZIONE VIENE INCREMENTATO DI UNO
- PER MIGLIORARE L'EFFICIENZA, QUESTO PROCESSORE FA IL PRE-FETCH. IL FETCH SAREBBE LA RACCOLTA DELL'ISTRUZIONE E POI LA ESEGUE. IL PRE-FETCH PRENDE L'ISTRUZIONE SUCCESSIVA MENTRE NE STA ESEGUENDO UN'ALTRA

MASTER COPY
Tel. 050 0312126
Cell. 388 9837745

ESEMPI DI ISTRUZIONI IN ASSEMBLER SI SCRIVE PRIMA IL DESTINATARIO E POI IL SORGENTE

- MOV R9, R2 // SPOSTA R2 IN R9 → IL VECCHIO CONTENUTO DI R9 VIENE PERSO ⇒ SCRITTURA È DISTRUTTIVA, LETTURA NON È DISTRUTTIVA
- ADD R9, R7 IN QUESTO PROCESSORE NON ESISTONO OPERAZIONE A 3 OPERANDI { A+B → A ; A+B → C SOLO IN C } ⇒ DISTRUTTIVA

MODI DI INDIRIZZAMENTO — TIPI DIVERSI DI SPECIFICARE GLI OPERANDI

- IMMEDIATO (PUÒ ESSERE SOLO SORGENTE) LDI R17, 37 // PRENDE 37 E LO METTE IN R17. L'INDIRIZZAMENTO IMMEDIATO HA A CHE FARE CON LE COSTANTI UGUALE AL DATO (IL 37 STA DA QUALCHE PARTE DENTRO L'ISTRUZIONE)
- DIRETTO (PUÒ ESSERE USATO ANCHE COME DESTINATARIO) LDS R17, 0x0050 // PRENDE IL CONTENUTO DELLA CELLA E LO METTE IN R17 (LA COSTANTE È UGUALE ALL'INDIRIZZO DEL DATO) → STS 0x0050, R17

LOAD → SI PRENDE UNA COSA E SI METTE IN UN REGISTRO
STORE → SI PRENDE UNA COSA DA UN REGISTRO E SI METTE IN UNA CELLA DI MEMORIA

- ADD = SOMMA 2 NUMERI A 8 bit — IN UNA SOMMA DI 2 NUMERI A 8 bit, NON È DETTO CHE IL RISULTATO SIA A 8 bit, MA C'È LA POSSIBILITÀ CHE SI GENERI RIPORTO (CARRY). MA IL PROCESSORE "PRENDE NOTA" DEL FATTO CHE C'È CARRY
- ADC = SOMMA 2 NUMERI E TIENE CONTO DEL CARRY DI UN'OPERAZIONE PRECEDENTE

ADC Rd, Rs // Rd ← Rd + Rs + CARRY

(ES: 0xC8 + 0xC8 → 1100 1000 + 1100 1000 = (1)1001 0000, CARRY)

PRIMA FACCIO UNA ADD CON GLI 8 bit MENO SIGNIFICATIVI E POI FACCIO UNA ADC CON QUELLI PIÙ SIGNIFICATIVI

SE VOGLIO SOMMARE 2 CIFRE A 16 bit MA L'HARDWARE MI PERMETTE DI FARE SOMME A 8 bit ⇒

- ADIW R4, 0x1234 // R5:R4 ← R5:R4 + 0x1234 → CIOÈ ADIW FA AUTOMATICAMENTE UNA SOMMA CON UN DATO IMMEDIATO A 16 bit E LO VA A METTERE NELLA COPPIA DI REGISTRI
- SUB
- SBC // PRESTITO (BORROW) → VIENE MESSO A 1 IL CARRY ⇒ SBC Rd, Rs // Rd ← Rd − Rs − CARRY
- SUBI → SOTTRAZIONE COL DATO IMMEDIATO (NON C'È ADDI POICHÉ BASTA FARE SUBI Rd, -k ⇒ HANNO RINUNCIATO A ADDI PER PROBLEMI DI SPAZIO)
- SBCI
- SBIW (ANALOGO A ADIW)
- AND (BIT A BIT)
- ANDI (AND CON UN DATO IMMEDIATO (COSTANTE) ⇒ SERVE PER VERIFICARE SE UN BIT È 1 O 0)

```
0 1 1 0 1 0 X 0 1   AND
0 0 0 0 0 0 1 0 0   ← MASCHERA
-----------------
0 0 0 0 0 0 X 0 0
```

- OR
- ORI
- EOR (OR ESCLUSIVO)
- COM ~ OPERAZIONE A UN REGISTRO ⇒ INVERTE IL REGISTRO
- NEG ~ CAMBIA DI SEGNO AD UN NUMERO IN COMPLEMENTO A 2
- INC (INCREMENTA ⇒ SOMMA 1 A UN REGISTRO)
- DEC (DECREMENTA ⇒ −1) ← UTILI PER I CICLI
- CLR (CANCELLA UN REGISTRO ⇒ TUTTI 0) ← FISICAMENTE NON ESISTE ~ LA SOSTITUISCE CLR Rd CON EOR Rd, Rd
- SER (SETTA UN REGISTRO ⇒ TUTTI 1)
- MUL Rd, Rr I DUE OPERANDI SONO ENTRAMBI SORGENTE E LA DESTINAZIONE È LA COPPIA R1:R0 // R1:R0 ← Rd × Rr (NON SI POSSONO METTERE R1 o R0 COME SORGENTE ALTRIMENTI IL RISULTATO È INDEFINITO)

- IL COMPLEMENTO A 2 È TRASPARENTE ⇒ L'HARDWARE DI UN SOMMATORE DI OPERANDI SENZA SEGNO O IN C2 È UGUALE. QUESTO È VERO ANCHE PER LA SOTTRAZIONE (LE RETI LOGICHE SONO UGUALI)

MA SE USO LA RETE LOGICA DI UN SOTTRATTORE O DI UN SOMMATORE PER OPERANDI IN C2, IL RISULTATO È ESATTO (IN COMPLEMENTO A 2) SE QUESTO È RAPPRESENTABILE

- QUESTA PROPRIETÀ DI TRASPARENZA NON VA BENE PER LA MOLTIPLICAZIONE

⇒ MULS CIOÈ MUL CHE TIENE CONTO DEL SEGNO ⇒ FA LA MOLTIPLICAZIONE DI OPERANDI IN COMPLEMENTO A 2
MA SI HA BISOGNO DI UNA TERZA ISTRUZIONE PER LA MOLTIPLICAZIONE "MISTA" ⇒ MULSU // MOLTIPLICA IL PRIMO OPERANDO COL SEGNO E IL SECONDO SENZA

- LA DIVISIONE NON C'È MA NON SAREBBE TRASPARENTE

- SREG [8 bit] ← DENTRO IL PROCESSORE C'È UN ALTRO REGISTRO SPECIALE: LO STATUS REGISTER

I	T	H	S	V	N	Z	C

- C = CARRY { RIPORTO IN UNA SOMMA ; PRESTITO DA UNA SOTTRAZIONE }
- Z = ZERO → È UGUALE A 1 OGNI VOLTA CHE L'ULTIMA OPERAZIONE ARITMETICA O LOGICA FA 0 (DOPO UN'OPERAZIONE SENZA SEGNO, ANALIZZANDO IL CARRY SI SCOPRE SE È RAPPRESENTABILE O NO) MA SE FACCIO OPERAZIONI COL SEGNO, QUESTA COSA NON FUNZIONA
- S, V, N SERVONO PER IL COMPLEMENTO A 2

ES: −1 → 0xFF (−1) + (−1) = −2 ⇒ SI PUÒ FARE PERCHÉ −2 È RAPPRESENTABILE

```
  0xFF  +
  0xFF
------
(1)0xFE = -2
```

CARRY ⇒ NEL C2 IL CARRY NON È SIGNIFICATIVO ⇒ S=1; V=0; N=1

- V = OVERFLOW ~ SE V=1 IL RISULTATO NON È RAPPRESENTABILE
- N = NEGATIVE ~ SEGNO EFFETTIVO DEL RISULTATO { N=1 SE <0 ; N=0 SE >0 }
- S = SIGN ~ SEGNO DEL RISULTATO SE FOSSE RAPPRESENTABILE { S=1 SE <0 ; S=0 SE >0 }

ES: 100 + 100 = 200 (NON È RAPPRESENTABILE)

```
  0x64  +
  0x64
------
  0xC8 → NON È 200 MA UN NUMERO < 0
```

CARRY = 0 ⇒ S=0; V=1; N=1

⇒ { V=0 → S=N ; V=1 → S≠N } ⇒ { V = S⊕N ; S = V⊕N ; N = V⊕S }

⇒ SE STO SOMMANDO NUMERI SENZA SEGNO → GUARDO C, SE INVECE SOMMO NUMERI IN C2 → GUARDO V

- H = HALF CARRY → IN UNA SOMMA A 8 bit, H È IL RIPORTO SUL 4° bit

IN MOLTE APPLICAZIONE SI UTILIZZA LA RAPPRESENTAZIONE BCD (BINARY CODED DECIMAL) E NELLA SOMMA DI DUE NUMERI BCD L'HALF CARRY È UTILE

① SOMMA ≤ 9 ⇒ OK
② 10 ≤ SOMMA ≤ 15 (RIPORTO MANCATO) ⇒ AGGIUNGO 6 (16 − 10)
③ SOMMA ≥ 16 (RIPORTO IN PIÙ) ⇒ AGGIUNGO 6

① 43 + 43 = 89 → ≤9 OK
② 46 + 47 = 8D → +6 = 93 → OK
③ 49 + 48 = 91 → +6 = 97 OK

CASO	CIFRA ACCETTABILE IN BCD	H
1	SÌ	0
2	NO	0
3	SÌ	1

MASTER COPY
Tel. 050 0312126
Cell. 388 9837745

SEP

Lezione 2

- LE ISTRUZIONE ARITMETICHE AGGIORNANO I FLAG DI SREG (Z=1 SE IL REGISTRO HA TUTTI 0, IN UNA SOMMA O SI FA 0+0 OPPURE 255+1 PERCHÈ VIENE 10...0 MA C=1)
  C, S, N, V SONO SEMPRE AGGIORNATI (E AGGIORNA ANCHE H).
  T = TEMPORARY STORAGE E ANCHE I NON VENGONO AGGIORNATI
- LE ISTRUZIONI LOGICHE AGGIORNANO SOLO Z (ANCHE N, V MA NON SONO SIGNIFICATIVI)
- LE MOLTIPLICAZIONI, CHE SONO IMPLEMENTATE DA UN'ALTRA RETE COMBINATORIA RISPETTO AL SOMMATORE, AGGIORNANO SOLO I FLAG Z E C (ANCHE SE NON CI PUÒ ESSERE CARRY)

ISTRUZIONI DI SALTO

SALTO INCONDIZIONATO → "JUMP" = SI ESEGUE UN'ALTRA ISTRUZIONE (NON QUELLA SUCCESSIVA) SENZA GUARDARE LA CONDIZIONE

INDIRIZZAMENTO RELATIVO. CIOÈ, DENTRO L'ISTRUZIONE NON C'È SCRITTO VAI ALL'INDIRIZZO TOT, MA C'È SCRITTO DI QUANTE CELLE ANDARE PIÙ AVANTI O INDIETRO

- RJMP CIOÈ: L'ARGOMENTO DI RJMP È UN NUMERO IN COMPLEMENTO A 2, MA NON È UN INDIRIZZO ASSOLUTO ⇒ VA AGGIUNTO AL PROGRAM COUNTER (ES: +3, -2)
  È UN NUMERO A 14 bit → QUINDI NON SI PUÒ SALTARE MOLTO LONTANO (NELL'ATMEGA 32 IN REALTÀ SI PUÒ ARRIVARE OVUNQUE POICHÈ BASTANO 14 bit PER LE CELLE IMPLEMENTATE)
- JMP NUMERO A 16 bit SENZA SEGNO → SALTO ASSOLUTO

QUESTE FANNO 3 COSE CONTEMPORANEAMENTE: ESEGUE L'ISTRUZIONE; INCREMENTA IL PROGRAM COUNTER; CARICA L'ISTRUZIONE SUCCESSIVA (AD OGNI CICLO DI CLOCK)

QUINDI, QUANDO SI HA UN SALTO IL PRE-FETCH, CHE PRENDE L'ISTRUZIONE SUCCESSIVA, VIENE BUTTATO VIA (CIOÈ IL PRE-FETCH FALLISCE) → ALLORA CI VUOLE UN CICLO DI CLOCK IN PIÙ (NEI PROCESSORI PIÙ POTENTI C'È IL "BRANCH PREDICTOR" CHE CERCA DI INDOVINARE DOVE VA IL FETCH)

QUANDO ESEGUE L'ISTRUZIONE, IL PROGRAM COUNTER È GIÀ STATO INCREMENTATO (QUINDI SE VOGLIO SALTARE IN UNA ISTRUZIONE, DEVO METTERE IL VALORE DELL'ISTRUZIONE PRIMA PERCHÈ IL PC È GIÀ STATO INCREMENTATO)

```
RJMP  k   // PC ← PC+1+k
RJMP  -1  // PC ← PC+1-1 → RIESEGUE IN CONTINUAZIONE (SI BLOCCA)
RJMP  0   // ESEGUE L'ISTRUZIONE SUCCESSIVA (SEGUE IL COMPORTAMENTO NORMALE DEL PROCESSORE)
```

} PRIMA DI SCRIVERE IL CODICE SI DEVE FARE L'ARITMETICA DEGLI INDIRIZZI. PER CUI SE NE OCCUPA IL PROGRAMMA AUTOMATICAMENTE

GLI ULTIMI 6 REGISTRI SONO CHIAMATI: X = {XL = R26, XH = R27} Y = {YL = R28, YH = R29} Z = {ZL = R30, ZH = R31}

- IJMP (X, Y, Z) → INDIRIZZAMENTO INDIRETTO A 16 bit
- CP R9, R7 → COMPROMETTA I DUE ARGOMENTI // FA COME SUB MA BUTTA VIA IL RISULTATO E AGGIORNA I FLAG ⇒ NON MODIFICA I REGISTRI
- CPC // "VIA" ← R9 - R7 - CARRY
- CPI R9, 12 // CP CON DATO IMMEDIATO

"COMPARE"	Z	C
R9 > R7	0	0
R9 < R7	0	1
R9 = R7	1	0
IMPOSSIBILE	1	1

SALTO CONDIZIONATO → BRANCH (SE NON SALTANO, ESEGUENDO L'ISTRUZIONE SUCCESSIVA) → INDIRIZZO A 6 bit ⇒ AL MAX {32 ISTRUZIONI AVANTI, 31 ISTRUZIONI INDIETRO}

- BRCC // SALTA SE C=0
- BRCS // SALTA SE C=1
- BRTC // SALTA SE T=0
- BRTS // SALTA SE T=1
- BRHC // SALTA SE H=0
- BRHS // SALTA SE H=1
- BRNE // SALTA SE Z=0 (NE = NOT EQUAL)
- BREQ // SALTA SE Z=1 (EQ = EQUAL)

} USATE SPESSO CON CP

HANNO UN INDIRIZZO DIRETTO RELATIVO (cz)
→ L'INDIRIZZO DI SALTO È PC+1+k

LE BRANCH SONO MOLTE DI PIÙ PERCHÈ SONO PRESENTI ALIAS (NO ES: BRLO // SALTA SE MINORE (SALTA SE C=1))

ISTRUZIONI DI TRASFERIMENTO DATI → PRENDONO UN NUMERO DA UN REGISTRO E LO METTONO DA ALTRE PARTI {REGISTRO, MEMORIA} [NON AGGIORNANO I FLAG]

- MOV // DA REGISTRO A REGISTRO
- MOVW // MOVE WORD → VUOLE DUE REGISTRI A 16 bit ⇒ QUINDI UNA COPPIA DI REGISTRI. SONO INDICATI IN MODO IMPLICITO: CIOÈ SE INDICO UN REGISTRO, PRENDE LA COPPIA DI REGISTRI FORMATA DAL SUCCESSIVO E QUELLO (R2 → (R3:R2))
  → È CONVENIENTE USARE REGISTRI PARI, COSÌ QUELLO PARI È IL MENO SIGNIFICATIVO
- LDI // PRENDE UN DATO IMMEDIATO E LO METTE NEL REGISTRO. CON LDI SI PUÒ USARE SOLO DA R16 A R31. E QUESTO VALE PER TUTTE LE IMMEDIATE
- LD → RAGGIUNGE TUTTI I REGISTRI. USA INDIRIZZAMENTO INDIRETTO.

```
LD R2, X   // IL REGISTRO X CONTIENE L'INDIRIZZO, VA IN MEMORIA (RAM), PRENDE IL CONTENUTO DELLA CELLA E LO METTE IN R2.
           L'INDIRIZZI NELLA RAM SONO A 16 bit, PER QUESTO CI VUOLE X, Y O Z (CHE SONO A 16 bit). LA MEMORIA HA UN DATO A 8 bit → LA DESTINAZIONE
           È UN REGISTRO A 8 bit
```

- LD R2, Y+q → CON LA LOAD POSSO FARE ANCHE POST-INCREMENTO; PRE-DECREMENTO {X+, -X}. INOLTRE POSSO AVERE INDIRIZZAMENTO INDIRETTO CON OFFSET (Y+q) DOVE q = COSTANTE A 6 bit SENZA SEGNO (Y NON VIENE MODIFICATO)
- ST // PRENDE DAL REGISTRO E LO METTE NELLA MEMORIA CON STESSE CONSIDERAZIONI DELLA LOAD

→ SE VOGLIO INCREMENTARE IL REGISTRO R8 ⇒ INC R8

→ SE VOGLIO INCREMENTARE UNA COPPIA DI REGISTRI R9:R8 → INCREMENTO R8 E SE HO CARRY INCREMENTO R9 ⇒ NON FUNZIONA PERCHÈ INC {AGGIORNA Z, MA NON C}

```
LDI R16, 1
CLR R17
ADD R8, R16
ADC R9, R17
```
⇒ OPPURE SI USA ADIW, MA SUBIW NON ESISTE

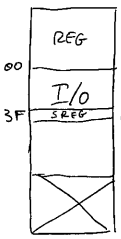
→ SE PERÒ I REGISTRI CHE DEVO INCREMENTARE SONO (X; Y; Z) ⇒ LDI R8, X+

- LDS R2, 0x0070 → LEGGE NELLA MEMORIA CON INDIRIZZAMENTO DIRETTO
- STS 0x0030, R3 → SCRIVE NELLA MEMORIA CON INDIRIZZAME DIRETTO

IN / OUT → ALLE CELLE CHE CONTROLLANO LE PERIFERICHE, È POSSIBILE ACCEDERE IN DUE MODI: {LDS, STS} CON GLI INDIRIZZI NEL RANGE 0x0020 ÷ 0x005F

OPPURE HANNO UN'ALTRA NUMERAZIONE CHIAMATA NUMERAZIONE DI I/O (O DI PERIFERICHE): CHE VA DA 00 A 3F

LA RELAZIONE SI OTTIENE PRENDENDO L'INDIRIZZO DI MEMORIA E TOGLIENDO 0x20 (32 IN DECIMALE)



- SE VOGLIO USARE QUESTI INDIRIZZI DEVO USARE DELLE ISTRUZIONI SPECIALIZZATE {IN // LEGGE, OUT // SCRIVE}

L'ULTIMA PERIFERICA (0x3F O 0x005F) È LO STATUS REGISTER → USATA PER AZZERARE O LEGGERE E SALVARE I FLAG

AZZERO I FLAG:
```
CLR R0
OUT 0x3F, R0
```

MASTER COPY
Tel. 050 8312126
Cell. 388 9837745

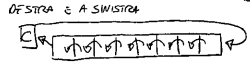
SEP

LEZIONE 3

- LE ISTRUZIONI SENZA OPERANDI CAMBIANO MANUALMENTE I FLAG ES: { CLC // CARRY=0 ; SEC // CARRY=1 ; CLZ // ZERO=0 ; SEZ // ZERO=1 }



- SHIFT → PRENDE UN REGISTRO E SPOSTA OGNI BIT NELLA POSIZIONE ADIACENTE. SI PUÒ FARE A DESTRA E A SINISTRA
- ROTATE → ANELLO A 9 bit, CIOÈ 8 bit DEL REGISTRO PIÙ IL CARRY { ROL // ROTATE LEFT ; ROR // ROTATE RIGHT }



ESISTONO DUE TIPI DI SHIFT: LOGICO e ARITMETICO

- LSL e LSR → LOGICO. SERVONO SE SI DEVONO SPOSTARE i BIT, OPPURE { SE SPOSTO TUTTI I BIT VERSO SINISTRA, AGGIUNGENDO UNO 0 ⇒ MOLTIPLICO PER 2 MA IL RISULTATO POTREBBE NON ESSERCI RAPPRESENTABILE (SIA CON CHE SENZA SEGNO) ; SE SHIFTO A DESTRA ⇒ DIVIDO PER 2 { SE IL NUMERO ERA PARI → C=0 ; SE IL NUMERO ERA DISPARI → C=1 } (IL RISULTATO È SEMPRE RAPPRESENTABILE) } DIVIDERE PER 2 NON FUNZIONA IN COMPLEMENTO A 2

ES: −2 IN COMPLEMENTO A 2 ⇒ 11111110 (0xFE)
↳ È PARI, E VOGLIO DIVIDERE PER 2 ⇒ SHIFTO A DESTRA ⇒ 01111111 ROC=0 MA QUESTO NUMERO È POSITIVO (≠ −1)

⇒ SI RISOLVE NON FACENDO ENTRARE 0 o 1, MA FACENDO ENTRARE COPIE DEL M.S.B ⇒ SHIFT ARITMETICI = ASR e ASL

[MA IN REALTÀ LSL e ASL SONO DELLE ALIAS PERCHÈ SONO LA STESSA COSA, QUINDI UNA ESISTE E L'ALTRA È UN'ALIAS. IN PIÙ LE ISTRUZIONI LSL e ASL SONO IN REALTÀ DELLE ADD DI UN REGISTRO CON SE STESSO]

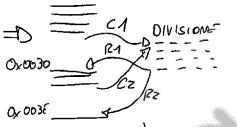
- SWAP SCAMBIA I DUE BLOCCHI DI 4 bit
- NOP // NO OPERATION = NON FA NIENTE, MA IL CICLO DI CLOCK LO FA COMUNQUE
- SLEEP // IL PROCESSORE VA IN "COMA" (DORME) → RISPARMIO ENERGIA (MA OCCORRE SVEGLIARLO)
- WDR // WATCH DOG RESET = DISPOSITIVO DI SICUREZZA (NON È ABILITATO DI DEFAULT, QUINDI VA ABILITATO OGNI TOT SECONDI)
- TST // TEST = GUARDA UN REGISTRO E AGGIORNA I FLAG DI CONSEGUENZA (EQUIVALE A FARE AND DI UN REGISTRO CON SE STESSO)

MASTER COPY
Tel. 050 8312126
Cell. 388 9837745

STACK → STRUTTURA DATI DOVE SI ACCUMULANO DATI TEMPORANEAMENTE [GESTITO SECONDO IL CRITERIO LIFO]
SERVE PER GESTIRE LA POSSIBILITÀ DI AVERE DEI SOTTOPROGRAMMI (AD ESEMPIO VOGLIO IMPLEMENTARE LA DIVISIONE)



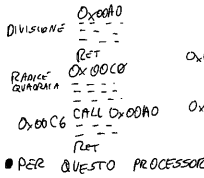
CONVIENE RICICLARE I PEZZI UGUALI ⇒

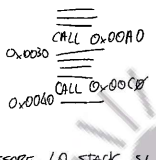


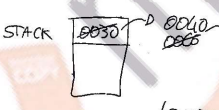
L'INDIRIZZO DI PARTENZA LO CONOSCO, QUINDI FACCIO UNA JUMP (O UNA RJUMP), MA IL PROBLEMA È SUL RITORNO PERCHÈ OGNI VOLTA È DIVERSO IL PUNTO DI PARTENZA
⇒ ALLORA SI USA UN'ISTRUZIONE CHE QUANDO PARTE, PRIMA DI FARE IL SALTO, SALVA NELLO STACK L'INDIRIZZO DI RITORNO
(L'ISTRUZIONE DI SALTO PRENDE L'INDIRIZZO DI RITORNO E LO METTE NELLO STACK ⇒ RCALL CALL
L'ISTRUZIONE DI RITORNO PRENDE L'INDIRIZZO DI RITORNO NELLO STACK ⇒ RET

NELLO STACK VENGONO CONSERVATI L'INDIRIZZI ASSOLUTI (16 bit)
LD PER IL RITORNO SI USA SEMPRE INDIRIZZI ASSOLUTI; PER LA CHIAMATA SI POSSONO USARE O L'INDIRIZZAMENTO RELATIVO OPPURE L'INDIRIZZO ASSOLUTO
(ALCUNI PROCESSORI USANO IL LINK REGISTER PER SALVARE L'INDIRIZZO DI RITORNO)

- LO STACK È UTILE PER LA GESTIONE DI SUBROUTINE NIDIFICATE: SUPPONIAMO DI DOVER IMPLEMENTARE LA RADICE QUADRATA CHE PER ESSERE CALCOLATA HA BISOGNO DELLA DIVISIONE

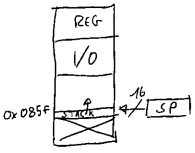






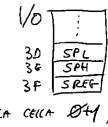
⇒ CI VUOLE UNO STACK PERCHÈ GLI INDIRIZZI DI RITORNO CI SERVONO NELL'ORDINE INVERSO IN CUI L'HO SALVATI
⇒ LA SUBROUTINE PIÙ SUBORDINATA È QUELLA CHE DEVE TORNARE PER PRIMA (L'ULTIMA CHE È ENTRATA È LA PRIMA CHE FINISCE)
(FUNZIONA LO STESSO CON SUBROUTINE CHE CHIAMANO SE STESSE)

- PER QUESTO PROCESSORE, LO STACK SI TROVA NELLA RAM. LO STACK CRESCE VERSO IL BASSO (PER CUI SI SCEGLIE UN INDIRIZZO PER LO STACK E MANO MANO CHE CRESCE, QUELL'INDIRIZZO DIMINUISCE). È CONSUETUDINE SCEGLIERE COME INDIRIZZO DI PARTENZA DELLO STACK, L'ULTIMO INDIRIZZO IMPLEMENTATO DALLA RAM



NELL'ESEMPIO, NELLO STACK VENGONO SALVATI INDIRIZZI DI RITORNO CHE SONO A 16 bit ⇒ 2 POSIZIONI. OGNI VOLTA CHE SI SALVA UN INDIRIZZO DI RITORNO, NON SI USA UNA CELLA DELLO STACK, MA DUE
(ES: SE FACCIO UNA CALL LO STACK POINTER DA 0x0B5F VA A 0x0B5D
SE FACCIO UNA RET LO STACK POINTER LO STACK POINTER SI INCREMENTA)

LO STACK POINTER È NELLO SPAZIO DI I/O
(AL PROGRAM COUNTER È INACCESSIBILE, ANCHÉ SE CON UNA JUMP SI PUÒ SCRIVERE NEL PC)



- [SE VOLESSI LEGGERE IL PC: RCALL 0 // SALTA ALL'ISTRUZIONE SUCCESSIVA (CHIAMATA A SUBROUTINE FINTA). PERCHÈ VA SALTA ALLA CELLA 0?, PERÒ MENTRE VA LÌ SALVA L'INDIRIZZO DI RITORNO, CHE SAREBBE QUELLO SOTTO LA CHIAMATA [VAI ALLA CELLA X E SALVA NELLO STACK L'INDIRIZZO X]

A QUESTO PUNTO NELLO STACK C'È L'INDIRIZZO DI RITORNO E PER PRENDERLO USO PUSH // METTE e POP // PRENDE, CHE PERMETTONO DI MANIPOLARE LO STACK
⇒ { RCALL 0 ; POP R0 ; POP R1 } LO DEVO METTERE IN DUE REGISTRI PERCHÈ A 16 bit (IN QUESTA MANIERA PERÒ NON SO QUALE REGISTRO CONTIENE I BIT PIÙ SIGNIFICATIVI E QUALE QUELLI MENO)
(SAREBBE MEGLIO METTERE NEI DISPARI LA PARTE PIÙ SIGNIFICATIVA)

- ALLA PARTENZA DEL PROCESSORE LO STACK POINTER NON È INIZIALIZZATO, PERCIÒ QUESTE ISTRUZIONI NON FUNZIONANO. QUINDI SI DEVE INIZIALIZZARE LO STACK POINTER, CIOÈ METTERE DENTRO SP UN INDIRIZZO VALIDO ⇒ { LDI R16, 0x5F ; LDI R17, 0x08 } SERVONO 2 REGISTRI PERCHÈ L'INDIRIZZO È A 16 bit ⇒ 0x085F È L'ULTIMO INDIRIZZO VALIDO, PER CUI METTO I BIT PIÙ SIGNIFICATIVI IN R17 E QUELLI MENO IN R16

{ OUT SPL, R16 ; OUT SPH, R17 } TRASFERISCO I REGISTRI NELLO STACK POINTER

ESISTONO DUE TIPOLOGIE PER PROGETTARE UN PROCESSORE: CISC e RISC (REDUCED INSTRUCTION SET COMPUTER). CONVIENE METTERE POCHE ISTRUZIONI MOLTO EFFICIENTI (UTILI). INOLTRE BISOGNEREBBE ARRIVARE ALL'OBIETTIVO CHE CIASCUNA ISTRUZIONE VENGA ESEGUITA IN UN CICLO DI CLOCK E OGNI ISTRUZIONE IN UNA SOLA CELLA DI MEMORIA.
I CISC AVEVA SPESSO ISTRUZIONI COMPLESSE CON DIVERSI CICLI DI CLOCK, PER CUI IL PROGRAMMATORE NON LI USA ⇒ ORA ORMAI SONO TUTTI RISC

I PROGETTATORI DI ATMEGA 32 SI VANTANO DI ESSERE MICROCONTROLLORI RISC, MA NON LO SONO DEL TUTTO: LE ISTRUZIONI DI SALTO NON FANNO TUTTO IN UN CICLO DI CLOCK (PER VIA DEL PRE-FETCH); RET E CALL NON SONO IN UN SOLO CICLO DI CLOCK PERCHÈ BISOGNA SALVARE L'INDIRIZZO DI RITORNO NELLO STACK (CHE RICHIEDE 2 CICLI DI CLOCK PER LA RET PERCHÈ BISOGNA ACCEDERE A DUE CELLA DI MEMORIA → 3 CICLI TOTALI)

MASTER COPY
Tel. 050 8312126
Cell. 388 9837745

SEP

LEZIONE 4

LE ISTRUZIONI HANNO SPESSO BISOGNO DI OPERANDI, UNO O DUE (SORGENTE e DESTINATARIO). CI SONO VARI MODI DI SPECIFICARE DOVE SONO GLI OPERANDI

MODI DI INDIRIZZAMENTO =
- **IMPLICITO** = QUANDO NON SERVE UN INDIRIZZAMENTO (ad esempio: MUL perché il destinatario è sempre R1:R0) oppure si usa anche per istruzioni che non hanno operandi (esempio: NOP)
- **IMMEDIATO** = QUANDO IL DATO È UNA COSTANTE CHE STA DENTRO L'ISTRUZIONE DA QUALCHE PARTE E NON PUÒ ESSERE MODIFICATA. PER QUESTO HA SENSO SOLO COME OPERANDO SORGENTE (#42)
- **REGISTRO** = L'OPERANDO STA IN UN REGISTRO, SIA PER SORGENTE CHE DESTINATARIO (ES: ADD)
- **DIRETTO** = NELL'ISTRUZIONE C'È UN NUMERO, CHE NON È UN DATO, MA L'INDIRIZZO DEL DATO (per distinguerlo dall'immediato alcuni processori usano un numero tra parentesi per il diretto (42) e il cancelletto per l'immediato)
- **INDIRETTO** = C'È UN REGISTRO CHE CONTIENE L'INDIRIZZO DEL DATO (ES: LD R3,Y // Y contiene l'indirizzo del dato. Se Y contenesse il dato ci sarebbe scritto YL e sarebbe un indirizzamento a registro)

FUNZIONAMENTO INTERNO DEL MICROCONTROLLORE cioè COME SONO CODIFICATE LE ISTRUZIONI (MAGGIOR INTERESSE PER CHI PROGETTA UN PROCESSORE)

LE ISTRUZIONI SONO DEI NUMERI A 16 bit CHE STANNO DENTRO LA FLASH

- ADD / ADC = `00 cy 11 rd dddd mmmm` (codice operativo operando destinazione 5 bit) — può cambiare — distanti → operando sorgente 5 bit (perché ci sono 32 registri); se C=0 → ADD, C=1 → ADC. CODICE OPERATIVO → 6 bit (serve per capire che istruzione va fatta. Tutti gli altri bit servono per l'indirizzamento)
- AND = `00 10 00 rd dddd mmmm` — HA IL CODICE OPERATIVO DIVERSO, MA HA I SOLITI BIT PER SPECIFICARE IL MODO DI INDIRIZZAMENTO, DIPENDONO QUINDI DAL REGISTRO
- OR = `00 10 10 rd dddd mmmm`
- NOP = 0000 0000 0000 0000

IL NUMERO DI BIT CHE CORRISPONDANO AL CODICE OPERATIVO E IL MODO D'INDIRIZZAMENTO PUÒ VARIARE

- BRANCH = `1111 0,B,SS ssss,sbbb` (codice operativo; indirizzo di destinazione dei salti in C2 e con indirizzamento diretto (salto relativo)) → B = SERVE A VEDERE SE L'ISTRUZIONE SALTA QUANDO IL FLAG È A 0 O QUANDO È A 1; → bbb = CI DICONO CHE FLAG È CONTROLLA SULLA CONDIZIONE DEL SALTO } CODICE OPERATIVO A 5 bit
- DEC = `1001 010d dddd 1010` (codice operativo; estensione del codice op.; operando destinazione) — ISTRUZIONE AD UN SOLO OPERANDO, PER CUI IL CODICE OPERATIVO È SPEZZATO IN DUE. PUSH e POP SONO SIMILI QUINDI BASTA CAMBIARE L'ESTENSIONE DEL CODICE OPERATIVO
- RJMP / RCALL = `110c kkkk kkkk kkkk` — RJMP e RCALL SONO SIMILI, HANNO UN OPERANDO 12 bit in C2 (indirizzo di salto relativo). C = DIFFERENZIA RJMP e RCALL → IL CODICE OPERATIVO È A 3 bit, perciò si può avere solo 8 istruzioni (a regola). IN REALTÀ NO, L'IMPORTANTE È CHE I CODICI OPERATIVI PIÙ LUNGHI NON COMINCINO CON QUEI 3 bit (110) (ES: A PISA → 050 → (2,4,3) → LA CENTRALE SI ASPETTA ALTRE 3 CIFRE, SE 050-5 → SI ASPETTA ALTRE 5 CIFRE)

PER QUESTO LDI DÀ PROBLEMI

- LDI R19, 37 → CI VOGLIO 8 bit PER SPECIFICARE IL DATO IMMEDIATO; 5 bit PER I REGISTRI (SONO 32), E ALLORA AVANZANO 3 bit PER IL CODICE OPERATIVO

MA DI TUTTI I CODICI OPERATIVI A 6 bit (64) NON POSSO USARNE 8, SE CONSIDERO CODICE OPERATIVO A 3 bit (QUELLI CHE INIZIANO 110)
→ CIOÈ OGNI CODICE A 3 bit MI TOGLIE LA POSSIBILITÀ DI USARE 8 COMBINAZIONI DI CODICI OPERATIVI PIÙ LUNGHI
QUINDI HANNO DOVUTO SACRIFICARE QUALCOSA PER FAR ENTRARE TUTTE LE ISTRUZIONI ⇒ PRIMA HANNO SACRIFICATO "ADDI" E POI PER RIENTRARE NEI BIT DISPONIBILI, PER LDI HANNO ACCORCIATO I BIT NECESSARI PER I REGISTRI, DA 5 A 4 PER CUI SOLO METÀ DEI REGISTRI È HANNO SCELTO QUELLA SUPERIORE. UNA DELLE d È UGUALE A 1
LDI = `1110 KKKK dddd KKKK` → LE 8 "K" CODIFICANO IL DATO IMMEDIATO E LE 4 "d" IL REGISTRO E IL CODICE OPERATIVO È A 4 bit (CHE È MENO COSTOSO PERCHÉ "MANCIA" SOLO 4 VALORI A 6 bit)

- LA SUBI e SBC HANNO CODICE OPERATIVO A 3 bit. ANCHE LE ISTRUZIONI CON INDIRIZZAMENTO INDIRETTO E NON CONSECUTIVI (10 KO)

SE IL DATO IMMEDIATO IMMEDIATO È A 16 bit, QUESTO MI OCCUPEREBBE TUTTO IL CODICE. ALLORA SONO STATI COSTRETTI A VENIRE MENO AL CONCETTO DI RISC IN CUI LE ISTRUZIONI SONO CONTENUTE IN UN'UNICA CELLA.

LDS R3, 0x1234 USA INDIRIZZAMENTO DIRETTO E C'È UN INDIRIZZO A 16 bit IN UNA CELLA DI MEMORIA SUCCESSIVA RISPETTO A QUELLA CHE CONTIENE LDS

- LDS / STS = `1001 00Sd dddd 0000` `KKKK KKKK KKKK KKKK` — LA PRIMA CELLA DI MEMORIA CONTIENE IL CODICE OPERATIVO (I PRIMI 6 bit), S DISTINGUE SE LDS o STS (DALLA MEMORIA AL REGISTRO O VICEVERSA). I 5 bit "d" CODIFICANO IL REGISTRO. I 4 bit 0 SONO I bit AVANZATI (QUANDO FANNO PARTE DEL CODICE OPERATIVO). LA LOCAZIONE SUCCESSIVA CONTIENE L'INDIRIZZO DIRETTO (DESTINAZIONE O SORGENTE)

QUESTO COMPORTA SICURAMENTE UN RALLENTAMENTO PERCHÉ OCCUPA 2 CICLI A CLOCK, COSÌ COME TUTTE LE ISTRUZIONI CHE HANNO UN DATO IMMEDIATO OPPURE UN INDIRIZZO DIRETTO A 16 bit
⇒ PER INDIRIZZARE LA MEMORIA, L'INDIRIZZAMENTO INDIRETTO È PIÙ EFFICIENTE DI QUELLO DIRETTO NEL NOSTRO PROCESSORE (SE DEVO INDIRIZZARE TANTE VOLTE UNA CELLA; SE INVECE DEVO ACCEDERCI UNA VOLTA OGNI TANTUM, CONVIENE QUELLO DIRETTO)

[O SI RINUNCIA AL RISC O SI RINUNCIA AL DATO IMMEDIATO]

ENDIANESS { BIG ENDIAN → SE DEVO DIRE QUARANTASETTE → DICO PRIMA IL PIÙ SIGNIFICATIVO E POI QUELLO MENO (BIG-ENDIAN = SI DICE PER PRIMA QUELLO GROSSO)
LITTLE ENDIAN → SUCCEDE IL CONTRARIO (VIENE PRIMA IL MENO SIGNIFICATIVO)

IN ELETTRONICA È MOLTO PIÙ COMUNE IL LITTLE-ENDIAN ⇒ SI TRASMETTE BIT A PARTIRE DAL MENO SIGNIFICATIVO (LSB) [IL NOSTRO PROCESSORE È LITTLE-ENDIAN]
(ALCUNI PROCESSORI HANNO DEI FLAG CHE POSSONO CAMBIARE L'ENDIANESS)

ALLINEAMENTO

PROCESSORE A 32 bit, MA COMUNQUE L'INDIRIZZI INDIANO IL SINGOLO BYTE (8 bit SONO MENO DI QUELLI CHE SERVONO, 32 bit SONO PIÙ DI QUELLI CHE SERVONO)

		H	L	0x0
H	M	M	L	0x4
	H	L		0x8
H	L			0xC
M	M	L		0x10
			H	0x14

- CON 8 bit, SE DEVO INSERIRE UN NUMERO A 16 bit, LO DEVO SPEZZARE E POI GESTIRE L'ENDIANESS
- CON 32 bit ABBIAMO IL PROBLEMA OPPOSTO E QUINDI DEVO RISPARMIARE ⇒ NEL REGISTRO A 16 bit → [H L] (LITTLE ENDIAN) ORA DEVE ANDARE IN MEMORIA: ALL'INDIRIZZO 0 ⇒ ALLINEATO oppure ALL'INDIRIZZO 1 ⇒ NON ALLINEATO MENTRE ALL'INDIRIZZO 2 È ALLINEATO PERCHÉ MULTIPLO DI 2 → ALL'INDIRIZZO 4 ⇒ ALLINEATO
- SE HO UN NUMERO A 32 bit [H M M L] → A PARTIRE DALL'INDIRIZZO 0x11 ⇒ NON ALLINEATO

⇒ SE L'INDIRIZZO IN CUI INIZIO A SCRIVERE È UN MULTIPLO DELLA LARGHEZZA DEL DATO, HO INDIRIZZAMENTO ALLINEATO (32 bit (4 Byte) → MULTIPLO DI 4)

L'HARDWARE ACCEDE A UNA RIGA DI 32 bit ALLA VOLTA ⇒ PER QUESTO QUANDO NON È ALLINEATO PUÒ CREARE PROBLEMI (ANCHE PERCHÉ COSTOSO A LIVELLO HARDWARE)

TIPOLOGIE DI PROCESSORI: (NON ALLINEATO SI o NO?)
- INTEL PERMETTE LA GESTIONE DELLA MEMORIA NON ALLINEATA, SOLO CHE IN TAL CASO L'OPERAZIONE È MENO EFFICIENTE (SI)
- (NO) CON AVVERTIMENTO → ALCUNI PROCESSORI PRODUCONO UN "FAULT" SE SI FA UN ACCESSO NON ALLINEATO
- (NO) SE SI PROVA A FARE UN ACCESSO NON ALLINEATO SI CREANO DEGLI ERRORI

⇒ PER EVITARE QUESTO PROBLEMA, SI ALLINEANO I DATI INSERENDO DEI BYTE VUOTI (SPRECO DI MEMORIA, MA TANTO È GRANDE)

MASTER COPY
Tel. 050 8312126
Cell. 388 9837745