ARCHITETTURA DEGLI ELABORATORI 2

L0 = Alu, registri, multiplexer;
L1 = Mic1;
L2 = binario;
L3 = solo funzionalità base, gestione memoria, I/O, file system, system call (procedure, funzioni) + linguaggio macchina;
L4 = simbolico;
L5 = programmi scritti in linguaggio ad alto livello;

MIC1 (L1)

CPU MIC1 = registri, unità aritmetico logica, controllo (microprogramma);
MAL = micro architetture language;
Ogni istruzione = 36 bit;
Cammino dei dati = quali dati devono essere forniti in input alla ALU e quali in output dalla **ALU** e shifter 3 Bus (Bbus, Cbus, Hbus);
MIR = micro instruction register;
MPC = micro program counter;
MDR = memory data register;

Input Enable = 1 quando è possibile leggere il dato del Cbus;

LINGUAGGIO SIMBOLICO MAL

operazione; lettura/scrittura verso la memoria; controllo di flusso

MAR = SP; rd; goto somma2

In un ciclo di clock
MAR, SP = registri
rd =read operazioni di memoria

OPERAZIONE
Source = uno degli ingressi sostituibile da:

SOURCE (bus B):
- MDR
- PC
- MBR
- MBRU
- SP
- LV
- CPP
- TOS
- OPC

(mai più di uno)

MAR non può essere caricato sul bus B;

DEST = indica dove mettere il risultato, si possono indicare anche più registri o anche 0;

- MAR = SP = SP + 1;

<< shift a sinistra

- H = MBR << 8; shift di 8 bit a sinistra e lo metto in H
- H = TOS = TOS >> 1; shift di TOS a destra di 1 bit e lo salvo in TOS e H;
- TOS = NOT SP; Ogni bit a 1 va a 0 e ogni bit a 0 va a 1 di SP e viene salvato in TOS;

SCRITTURA
rd, wr, fetch (può essere assente)

CONTROLLO DI FLUSSO

- goto label → salto incondizionato
- goto (MBR) → salto a molte vie
- goto (MBR or 0x100) → salto a molte vie
- if (Z) goto label1; else goto label2 → salto condizionato per risultato ALU == 0
- if (N) goto label1; else goto label2 → salto condizionato per risultato ALU < 0
- se è assente si assume che next_address sia la successiva istruzione nella sequenza

Se Z è 0 si salta a label1 se 1 si salta a label2
Se N è 1 si salta a label1 se 0 si salta a label2

Operazioni non ammissibili
MAR = MAR + 1 non può essere una SOURCE
MDR = MDR + SP nelle operazioni con due operandi uno dei due deve essere per forza H o una costante;
H = H - MDR (MDR - H va bene) H non può essere sottratto;
MAR = SP; rd; la lettura impiega un ciclo per restituire il valore letto;
MDR = H; riceverebbe simultaneamente dati dalla memoria e dal bus C;

N = espressione1; if (N) goto label1; else goto label2;

Z = espressione2; if (Z) goto label3; else goto label1;

La stessa etichetta non può trovarsi sia nel ramo if e nel ramo else;

```
Z = TOS; if (Z) goto fine; else goto somma1
    0   0    9    1    4    0    0    0    7
000000001001 0001 0100 0000 0000 0000 0111
```

Next-Address JAM Sh ALU C Mem B

B bus registers

0 = MDR	5 = LV
1 = PC	6 = CPP
2 = MBR	7 = TOS
3 = MBRU	8 = OPC
4 = SP	9-15 none

B = TOS; 7 in esadecimale;
Mem = rd, wr, fetch;
C = registri di destinazione (H, MAR, MDR);
ALU =

F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\overline{A}$
1	0	1	1	0	0	$\overline{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B - A
1	1	0	1	1	1	B - 1
1	1	1	0	1	1	-A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0	1	0	0	1	0	-1

JAM = controllo;
Next Address = indirizzo somma1;

```
istr0 Z = OPC = TOS + H; if (Z) goto istr1; else goto istr2
        0x0093C4007 →  0000 0000 1001 0011 1100 0100 0000 0000 0111
                       000000001 001 00 111100 010000000 000 0111
```

MIR avrà come valore 0x0093C3007
Decodifica campo bus B 0111→TOS
H→Bus A, TOS→ Bus B
ALU = A+B
Caricamente del risultato sul bus C e impostazione dei bit Z, N
Bus C → OPC, nessuna richiesta di accessi alla memoria
Settato il nuovo valore di MPC; salto condizionato bit più significativo a 0; salto multi via di solito tutti i bit non significativi a 0;

MAL è anche un linguaggio assemblativo
Ha bisogno di alcune informazioni (Ex. .label inizio 0x0, .default goto fine)

OPC=1010
^ negativo aggiungo 1;
Sommare il numero per se stesso equivale ad uno shift;
OPC+OPC non valido;
H=OPC;
OPC+H;
Mic1 non guarda l'overflow;

ARM = risc
Intel = cisc

MACROARCHITETTURA IJVM

Si usa lo stack perché è l'unico modo per gestire le chiamate ricorsive;
LV punta alla base del record corrente;
SP punta alla testa dello stack;

Il return svuota il record di attivazione;
Quando termino una chiamata devo gestire quale istruzione eseguire e quale record di attivazione diventa quello corrente;
1- indirizzo del chiamante;
2- indirizzo della funzione che devo eseguire quando termina la funzione chiamata.

MEMORIA (IJVM)
Il modello della memoria si divide in ¾ parti;

1) Metod area: strutturata a byte (Registro PC: punta alla prossima istruzione da eseguire)
2) Constant pool: (Registro CPP: punta alla base di quest'area)
3) Stack: struttura dinamica (Due registri: LV punta alla base del record corrente, SP punta alla testa dello stack)

PC puntatore a byte;
SP, LV, CPP puntatori a parole;

Per leggere il byte successivo dobbiamo usare il fetch;
Per accedere ai dati (accesso a parole) useremo wr, rd.

BIPUSH: carica un valore contenuto nel PC sopra lo stack;
LDC_W: scrive una costante sopra lo stack;
ILOAD: caricare il valore di una variabile nel record di attivazione corrente e caricarlo in cima allo stack;

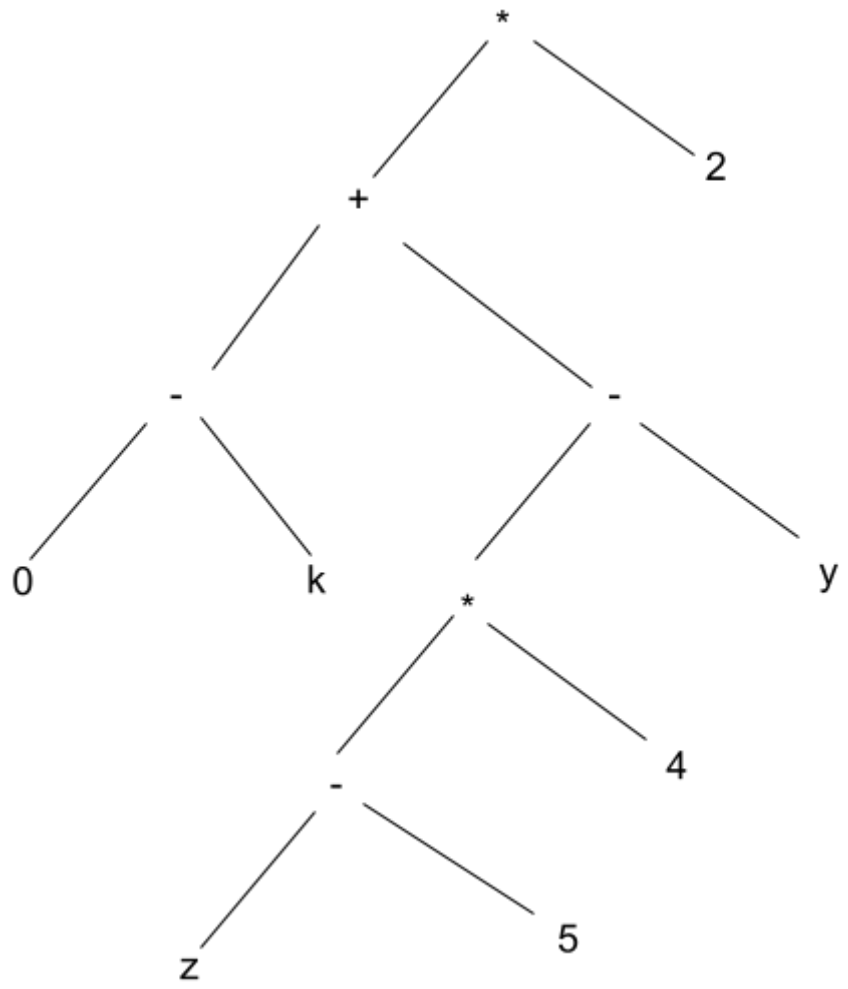
ISTORE: prendere il valore in cima allo stack e salvarlo in una variabile;
IADD: somma i due valori in cima allo stack sostituisce al secondo il risultato;
ISUB: sottrae il **penultimo** valore con l'**ultimo** valore;
DUP: duplica il valore in cima allo stack;
SWAP: inverte i due valori in cima allo stack;
GOTO: i due operandi si trovano nelle posizioni successive al goto, avremo un salto dove il PC si sposterà;
IFEQ: ci permette di realizzare un salto condizionale, saltiamo se in cima allo stack c'è il valore 0 e non saltiamo se in cima allo stack non c'è 0;
WIDE ILOAD: se ho bisogno di indicare un indice che richiede più di 8 byte;

Estensione del segno: se il numero è negativo devo copiare nei byte 2, 3, 4 il segno del byte che vado a caricare (0 positivo, 1 negativo);

REALIZZAZIONE DI IJVM
Invarianti = proprietà che deve valere sempre all'inizio del ciclo;

JAS (java assembler)

[- K + (z-5) * 4 – y] * 2



HALT = system call, fine del programma. (Non è obbligatorio)
La codifica IJVM è Big Endian.

Esercizio 2 (0-7 punti)
Dato il seguente codice IJVM/ JAL

```
.method max (p1, p2)
        ILOAD p1
        ILOAD p2
        ISUB
        IFLT maxP2
        ILOAD p1
        IRETURN
maxP2:  ILOAD p2
        IRETURN
.end-method

.constant
        OBJREF 0x40
        K 10
.end-constant

.main
.var
    n
    m
.end-var

BIPUSH 13
ISTORE n
BIPUSH 10
ILOAD n
ISUB
ISTORE m
LDC_W OBJREF
ILOAD m
LDC_W K
DUP

IADD
IADD
ILOAD n
BIPUSH 5
ISUB
INVOKEVIRTUAL max
ISTORE n
IINC m 1

.end-main
```

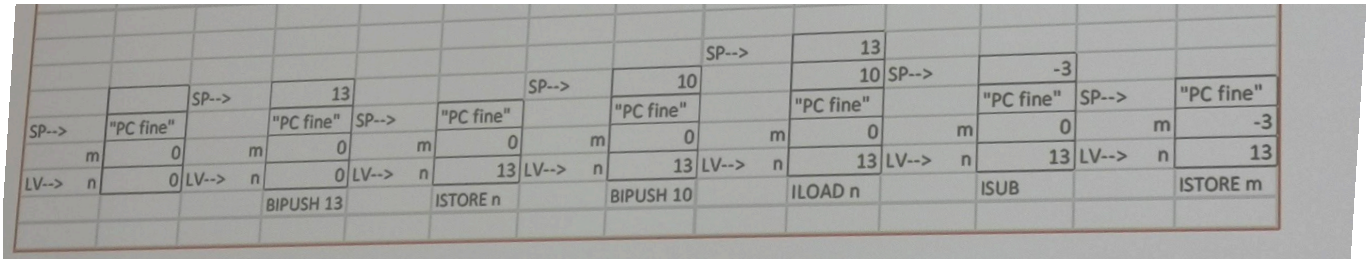
(a) Riprodurre il codice IJVM/JAL in (pseudo-)codice C
(b) Illustrate l'evoluzione dello stack durante l'esecuzione del main, partendo dalla configurazione iniziale dell'emulatore IJVM riprodotta qui di lato, indicando sullo stack anche i record di attivazione e le istruzioni del metodo max

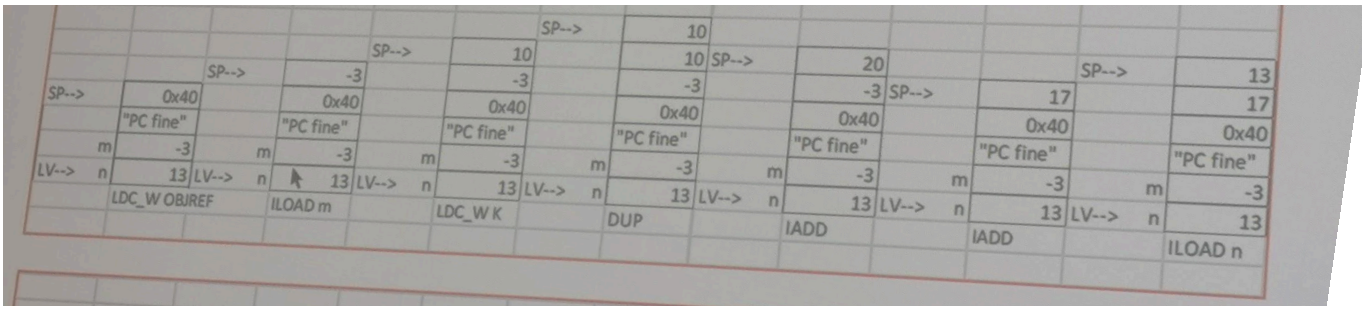
SP-->	"PC fine"
(m)	0
LV--> (n)	0

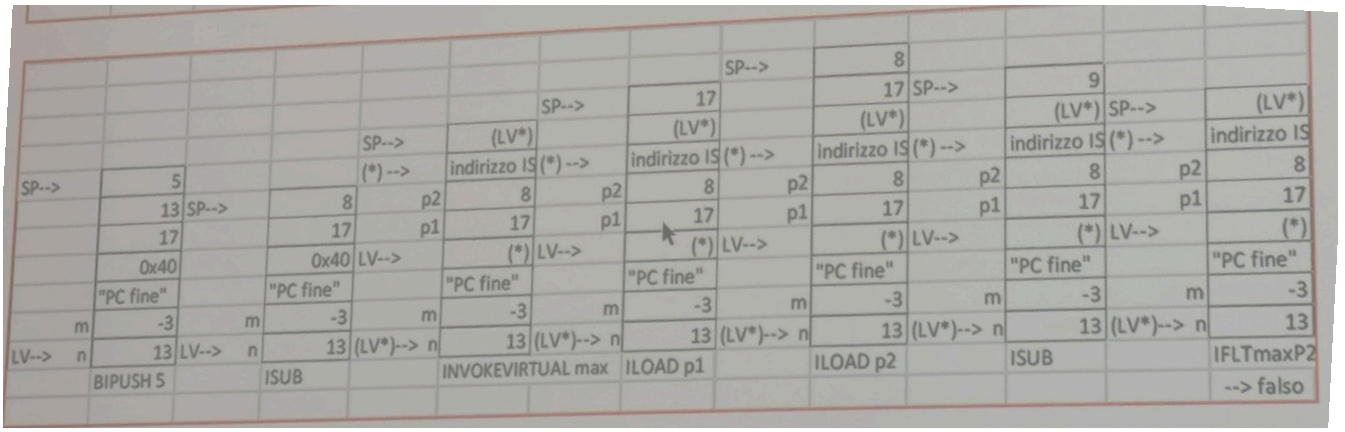
(a) Riprodurre il codice IJVM/JAL in (pseudo-)codice C

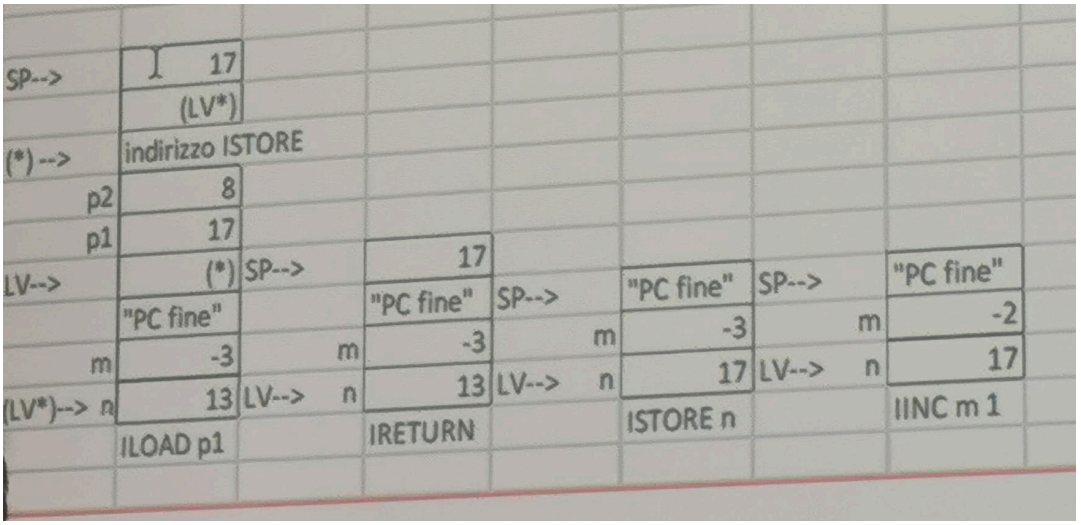
```
int max(int pi1, int p2) {
    if (p1<p2) {            //oppure if (p1-p2<0)
        return p2;
    } else {
        return p1:
    }
}

const K = 10;
int main() {
    int n =13;
    int m = 10-n;
    n = max(m+2*K, n-5);
    m++;
}
```

(b) Il valore finale delle variabili dopo l'esecuzione del main è:
n = 17
m = -2




SP--> 0x40 "PC fine" m -3 LV--> n 13 LDC_W OBJREF
SP--> -3 0x40 "PC fine" m -3 LV--> n 13 ILOAD m
SP--> 10 -3 0x40 "PC fine" m -3 LV--> n 13 LDC_W K
SP--> 10 10 -3 0x40 "PC fine" m -3 LV--> n 13 DUP
SP--> 20 -3 0x40 "PC fine" m -3 LV--> n 13 IADD
SP--> 17 0x40 "PC fine" m -3 LV--> n 13 IADD
SP--> 13 17 0x40 "PC fine" m -3 LV--> n 13 ILOAD n


SP--> 5 13 17 0x40 "PC fine" m -3 LV--> n 13 BIPUSH 5
SP--> 8 17 0x40 "PC fine" m -3 LV--> n 13 ISUB
SP--> (*) --> (LV*) indirizzo IS 8 p2 17 p1 (*) LV--> "PC fine" m -3 (LV*)--> n 13 INVOKEVIRTUAL max
SP--> 17 (LV*) indirizzo IS 8 p2 17 p1 (*) LV--> "PC fine" m -3 (LV*)--> n 13 ILOAD p1
SP--> 8 17 (LV*) indirizzo IS (*) --> 8 p2 17 p1 (*) LV--> "PC fine" m -3 (LV*)--> n 13 ILOAD p2
SP--> 9 (LV*) indirizzo IS (*) --> 8 p2 17 p1 (*) LV--> "PC fine" m -3 (LV*)--> n 13 ISUB
SP--> (LV*) indirizzo IS 8 17 (*) "PC fine" -3 13 IFLTmaxP2 --> falso


SP--> 17 (LV*) (*) --> indirizzo ISTORE p2 8 p1 17 LV--> (*) "PC fine" m -3 (LV*)--> n 13 ILOAD p1
SP--> 17 "PC fine" m -3 LV--> n 13 IRETURN
SP--> "PC fine" m -3 LV--> n 17 ISTORE n
SP--> "PC fine" m -2 LV--> n 17 IINC m 1

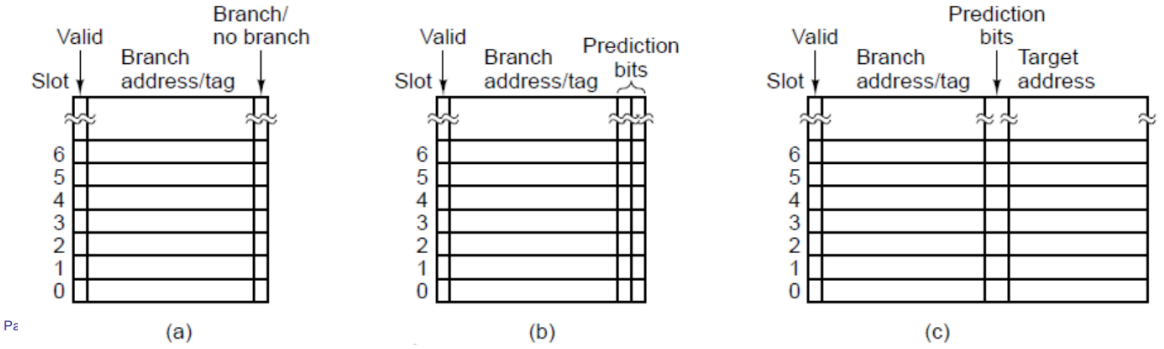
(b) Eseguire il codice di ILOAD sulla architettura MIC-3, indicando eventuali dipendenze.

	iload1	iload2	iload3
Clock	MAR = LV+MBR1U; rd	MAR = SP = SP + 1	TOS = MDR; wr; goto(MBR1)
1	A=MBR1U B=LV		
2	C=A+B	B=SP	
3	MAR=C; rd	C=B+1	bolla per RAW MDR
4	MDR←mem	MAR=SP=C	bolla per RAW MDR
5			B=MDR
6			C=B
7			TOS=C; wr
8			mem←MDR; goto(MBR1)

MBR1 e MBR2 non possono essere caricati sul bus B

Previsione dinamica

- Le informazioni necessarie per la predizione dinamica dei salti vengono memorizzate in **memorie associative** (memorie simili a quelle utilizzate per realizzare le cache) che sono indirizzate in base al contenuto
- Si chiede alla memoria associativa di restituire il contenuto del campo
  a) "Branch/no branch" (caso di predizione a 1 bit)
  b) "prediction bits" (caso di predizione a 2 bit)
  c) "Target address", nel caso in cui fosse calcolato dinamicamente
- in base al contenuto del campo "Branch address" (cioè in base all'indirizzo dell'istruzione di salto condizionato su cui dobbiamo effettuare la predizione)



All'inizio dell'esecuzione tutti i bit di validità vengono messi a 0, quando le istruzioni vengono caricate nella tabella i bit relativi a queste istruzioni vengono messi a 1.

RAW è una true dependence.
WAR e WAW possono essere raggirati tramite registri segreti.

CACHE

Cache livello 1 vicino alla CPU divisa in cache per istruzione e cache per i dati perché la memoria delle istruzione non deve essere modificata invece quella dei dati è accessibile in lettura e scrittura.
I bit possono essere ridondanti su più cache.
Più ci si avvicina alla CPU più potrebbe aumentare la velocità e ridurre lo spazio di archiviazione.
La RAM è suddivisa in blocchi, questi possono essere memorizzati nelle cache. Le cache memorizzano i blocchi nelle linee di cache.

tc tempo medio di accesso ad un dato in cache
tm tempo medio di accesso ad un dato in memoria centrale
h probabilità di accedere ad un dato presente in cache
m probabilità di non trovare il dato

E(t) tempo medio di accesso alla memoria
E(t) = tc + (1h) tm

Num blocchi = dimensione memoria / dimensione del Blocco

Numero del blocco = Indirizzo / dimensione del Blocco
offset distanza dell'indirizzo dalla base del blocco
offset = Indirizzo % dimensione del Blocco
Dimensione Blocco = 32b
Indirizzo = 42
Numero del Blocco = 42/32 = 1
offset = 42%32 = 10

Ogni riga contiene un blocco.
Bit di Validità dice se in quella linea è presente un dato rilevante per il programma
Tag per capire quale elemento è stato memorizzato in quel blocco di memoria (porzione dell'indirizzo).

Cache a corrispondenza diretta = ad ogni riga è associato un singolo blocco.

- Il quoziente della divisione (intera) N / 2^k si ottiene “eliminando” gli ultimi k bit (shift a destra di k bit)
- Il resto (o modulo) N % 2^k si ottiene “selezionando” gli ultimi k bit

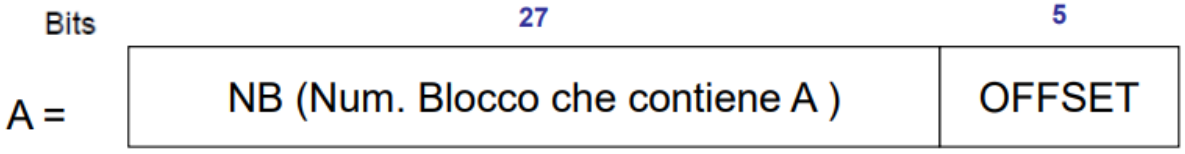
ESEMPIO

N = 1110 0000 1101;

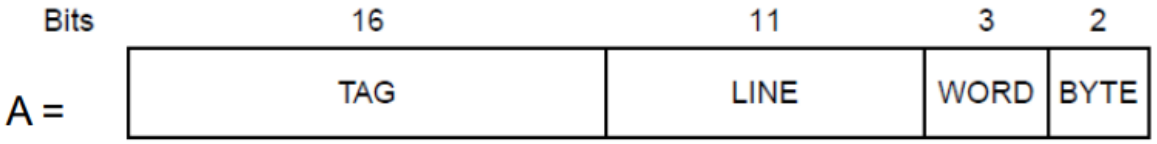
Se k = 4:

N / 2^4 = 1110 0000
N % 2^4 = 1101

Dimensione della memoria / Dimensione del Blocco = Numero di blocchi
2^32/2^5 = 2^27



Numero del Blocco%Numero di linee della cache = Dove salvare/cercare il blocco(Line)
I bit restanti si chiamano TAG



^NB ^offset

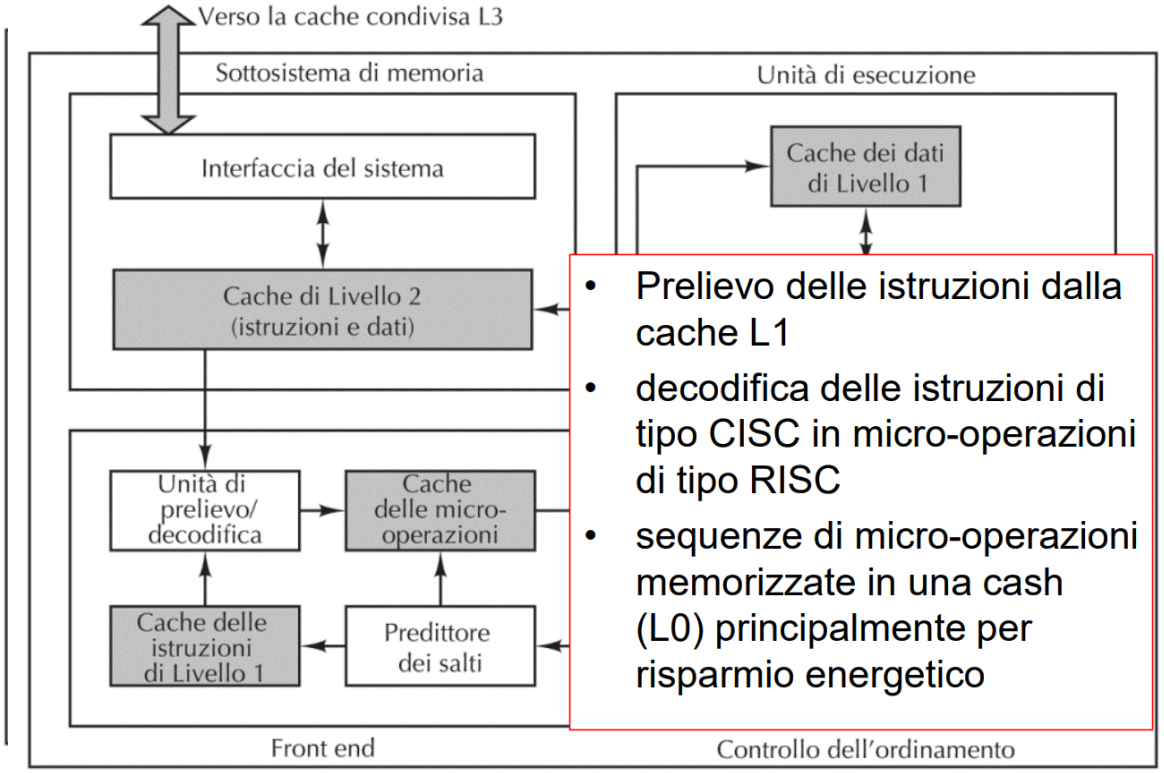
Due blocchi con la stessa LINE ma TAG diversi sono in collisione
Indirizzi distanti multipli di 65 Kbyte sono in collisione
Se hanno numeri di blocco multipli di 2^11 sono in collisione

Overhead rapporto delle informazioni della cache rispetto ai dati utili.

Cache set-associative a 4 vie riduce la possibilità di collisioni.

Core i7
Per riuscire ad eseguire le istruzioni CISC velocemente l’i7 ha un nucleo RISC strutturato a pipeline.
Cache L1, L2 del chip ed L3 condivisa.
L1-I: cache primo livello per le istruzioni.
L1-D: cache primo livello dei dati.
L2: cache unica per istruzioni e dati.

PIPELINE:



Cache di livello 0 equivale alla coda di Mic4

ARM è un progettista di chip, rende la specifica pubblica in modo che più costruttori possano implementare un'architettura.

OMAP4430
Implementazione di architettura ARM
Due livelli di cache

ARM è una macchina di tipo RISC non ha bisogno di decodifiche.

IJVM è il linguaggio macchina.
Per ottimizzare l'esecuzione bisogna conoscere l'architettura, (quante istruzioni vengono eseguite in parallelo, quanto è il blocco della cache).

Non tutte le istruzioni possono essere eseguite da tutti i programmi.

L'eccezione o trap potrebbe passare da user mode a kernel mode per poi tornare alla user mode una volta finita l'esecuzione dell'istruzione.

- parole di 4 byte allineate agli indirizzi multipli di 4 (0,4,8, ...), che terminano con 00;
- parole di 8 byte allineate agli indirizzi multipli di 8 (0,8,16,...), che terminano con 000;

00 prima cella 01 seconda e così via.

- *indirizzamento diretto*
  - l'operando è all'indirizzo di memoria specificato nell'istruzione
    LOAD R1, 0x102930 semantica: R1 ← m[0x102930]
  - accesso ad una locazione fissa della memoria
  - utile per accedere a variabili globali del programma

Solitamente per variabili globali o costanti.

Indirizzamento a segmento: un programma può essere strutturato su più segmenti (variabili, costanti ecc.). Ogni segmento è caratterizzato da un numero e vi è un offset.

Modalità di indirizzamento nel Core i7

	MOD			
R/M	00	01	10	11
000	M[EAX]	M[EAX + OFFSET8]	M[EAX + OFFSET32]	EAX or AL
001	M[ECX]	M[ECX + OFFSET8]	M[ECX + OFFSET32]	ECX or CL
010	M[EDX]	M[EDX + OFFSET8]	M[EDX + OFFSET32]	EDX or DL
011	M[EBX]	M[EBX + OFFSET8]	M[EBX + OFFSET32]	EBX or BL
100	SIB	SIB with OFFSET8	SIB with OFFSET32	ESP or AH
101	Direct	M[EBP + OFFSET8]	M[EBP + OFFSET32]	EBP or CH
110	M[ESI]	M[ESI + OFFSET8]	M[ESI + OFFSET32]	ESI or DH
111	M[EDI]	M[EDI + OFFSET8]	M[EDI + OFFSET32]	EDI or BH

- non tutte le modalità si applicano a tutti i registri → ESP non si può usare come base di un offset
- MOD = 11: il primo registro usato per le istruzioni che operano a parola, e l'altro per le istruzioni che operano a byte

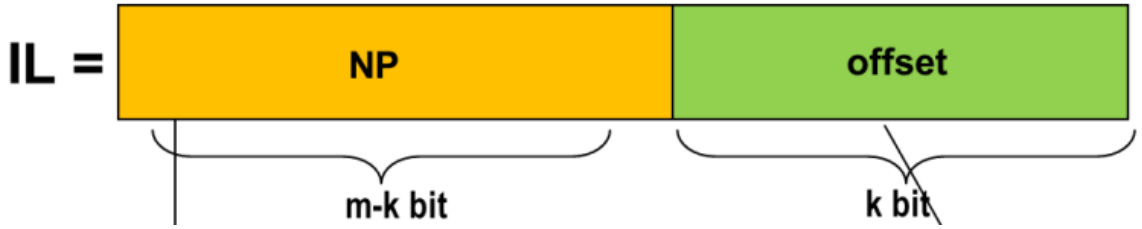
NON RICHIESTA ALL'ESAME

Branch with link = viene fatto il salto e viene salvato l'indirizzo nel registro in modo da saltare a quel valore di ritorno.

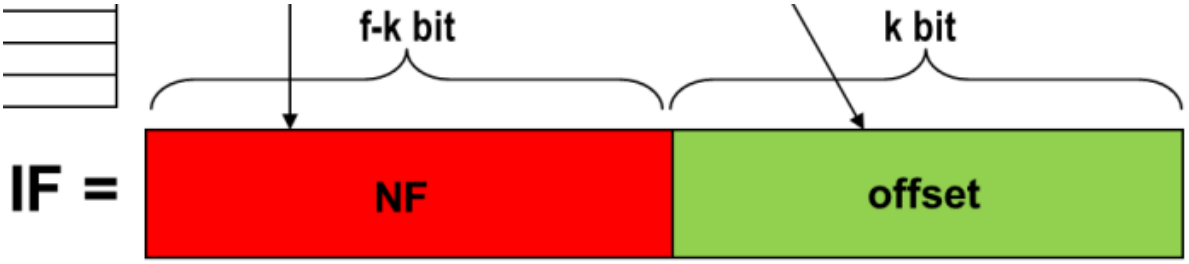
MEMORIA VIRTUALE
Spazio degli indirizzi:
Memoria fisica: indirizzato al singolo byte

Memoria paginata: la memoria viene suddivisa in blocchi
RAM -> memoria secondaria: swap out
Memoria secondaria -> RAM: swap in

Offset all'interno del blocco = Indirizzi logici % Dimensione Pagina
Numero pagina logica = Indirizzi logici / Dimensione Pagina
Indirizzo logico = Dimensione pagina * numero pagina + Offset



Indirizzo fisico = Dimensione pagina * numero pagina fisica + Offset



Tempo medio di accesso = (1 - frequenza page fault) * tempo di accesso alla RAM + frequenza page fault * tempo di gestione di un page fault

STOP SLIDE 52

Portare figure cap. 4
Portare fogli protocollo!
Sì calcolatrice

MAL: programmazione
binaria->simbolica o viceversa

IJVM: programmazione/estensione interprete/ esecuzione sullo stack
domande codifica binaria procedure
come funziona il record di attivazione

CACHE: esercizi

ISA: Indirizzamenti/Istruzioni di input/output/load, store/codice a espressione/ ARM

MICx: come sono le istruzioni di mic1 rispetto a mic2/pipeline/codifica istruzioni mic4/ predizione dei salti/ architetture superscalari/ dipendenze delle variabili

No domande ultimo argomento al primo appello