



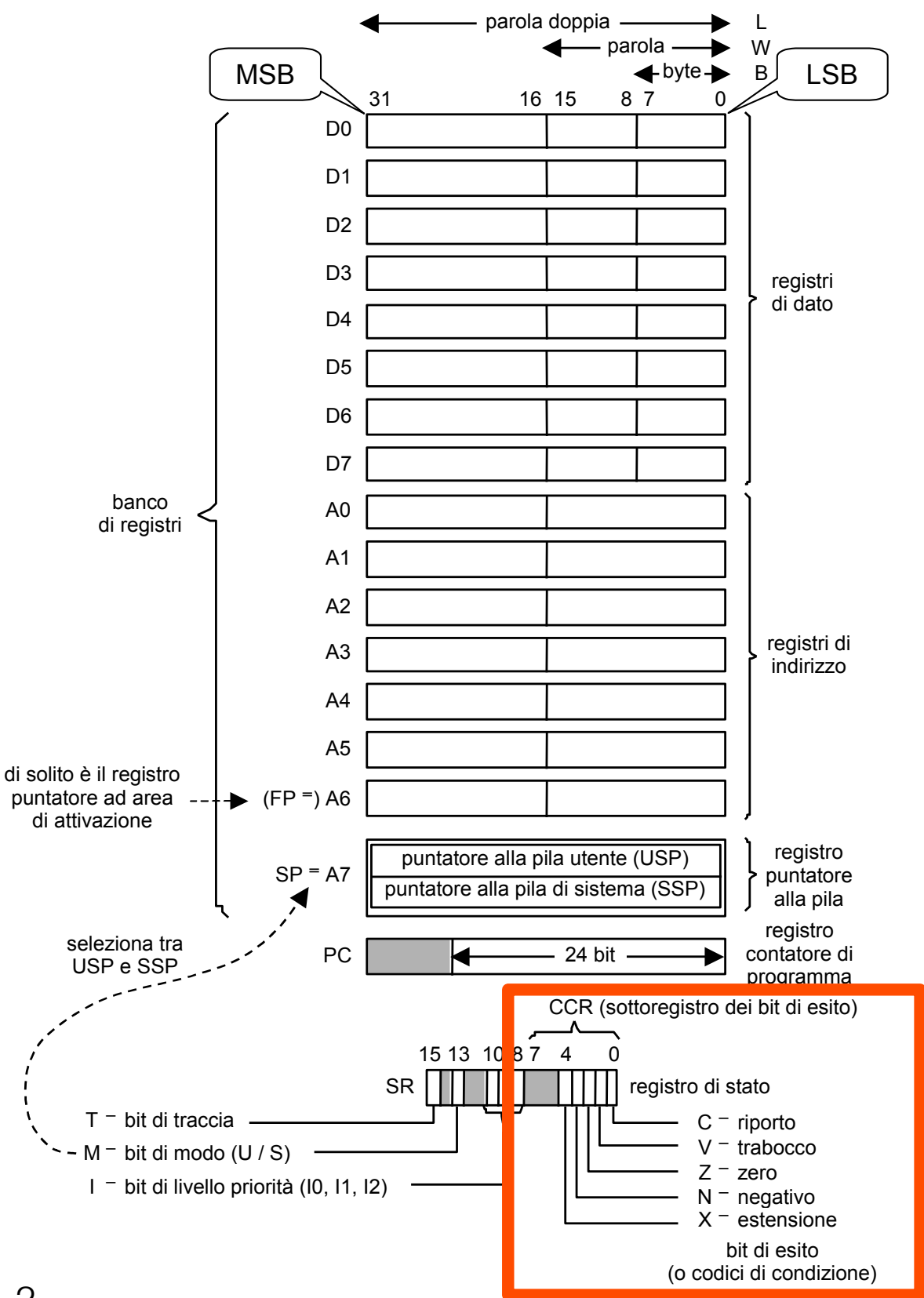
Esercizi Assembly

Alessandro A. Nacci
alessandro.nacci@polimi.it

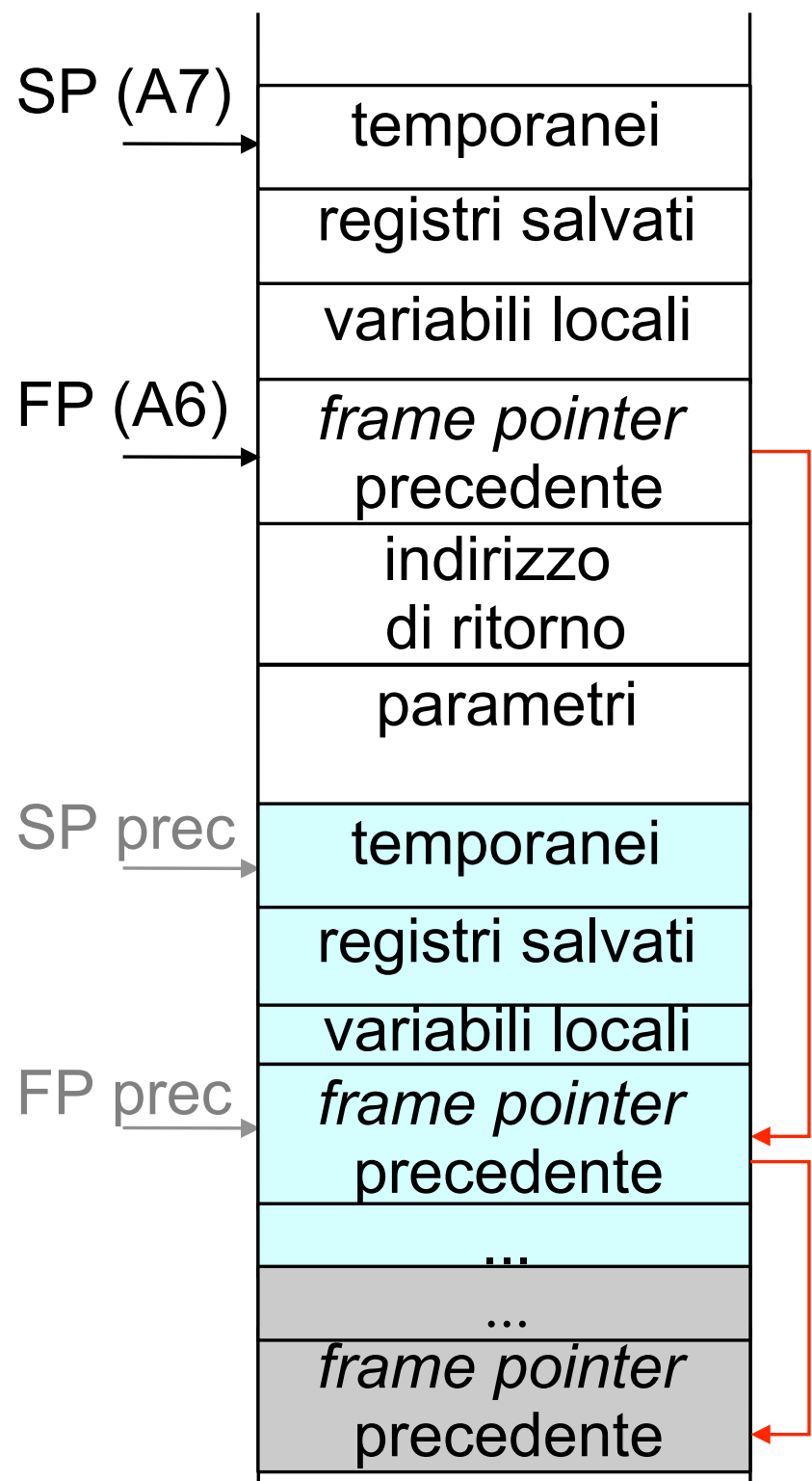
ACSO
2014/2014

Architettura dei registri del 68K

- Di** ($0 \leq i \leq 7$): registri di dato a 8, 16, 32 bit
- Ai** ($0 \leq i \leq 7$): registri di indirizzo a 16, 32 bit
- PC**: Program Counter a 32 bit (24 bit)
- SP** (= A7): Stack Pointer a 16, 32 bit
- FP** (di solito = A6): Frame Pointer
(puntatore all'area di attivazione) a 16, 32 bit
- CCR**: Condition Code Register (registro dei bit di esito - flag) a 8 bit
- SR**: Status Register
registro di stato (contiene CCR) a 16 bit



Sottoprogrammi: area di attivazione

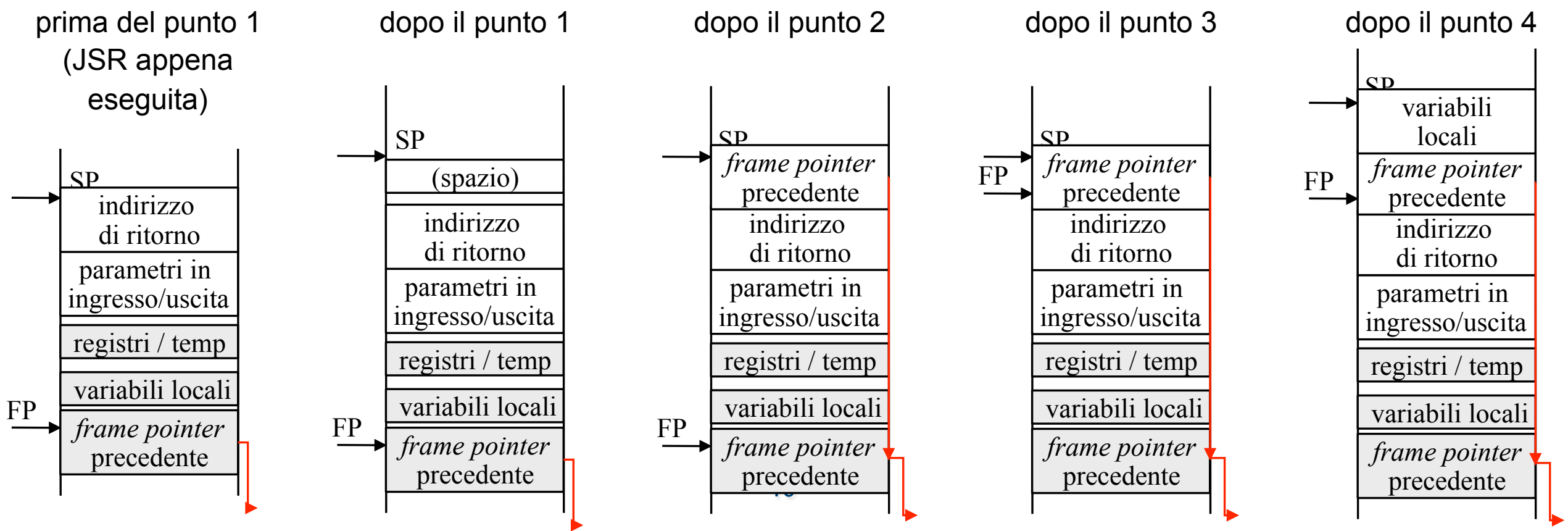


- una migliore realizzazione del meccanismo di chiamata a sottoprogramma fa uso delle **aree di attivazione** (*stack frame*)
- in 68000 esistono istruzioni dedicate per allocare e deallocare aree di attivazione: **LINK** e **UNLK**
- le aree di attivazione formano una lista concatenata (in rosso nella figura)
- per convenzione, il puntatore all'area di attivazione corrente (*frame pointer*) è nel **registro A6**
- creazione del frame **LINK A6, #-N**
(N è la dimensione delle variabili locali del chiamato)
- deallocazione del frame **UNLK A6**
- i **parametri** hanno spiazzamento **positivo rispetto a FP**
esempio: il primo parametro .L sta a **8 (A6)**
- le **variabili locali** hanno spiazzamento **negativo rispetto a FP**
esempio: la prima var. locale .L sta a **-4 (A6)**

Sottoprogrammi

LINK

- istruzione LINK A6, #-N**
 - $[SP] \leftarrow [SP] - 4$
riservo una parola lunga sulla pila
 - $[M([SP])] \leftarrow [A6]$
salvo vecchio *frame pointer*
 - $[A6] \leftarrow [SP]$
il *frame pointer* ora punta alla cima della pila
 - $[SP] \leftarrow [SP] + N$
riservo spazio per le variabili locali
(attenzione: -N è solitamente negativo)



Sottoprogrammi

UNLNK

□ **istruzione UNLK A6**

1. [SP] ← [A6]

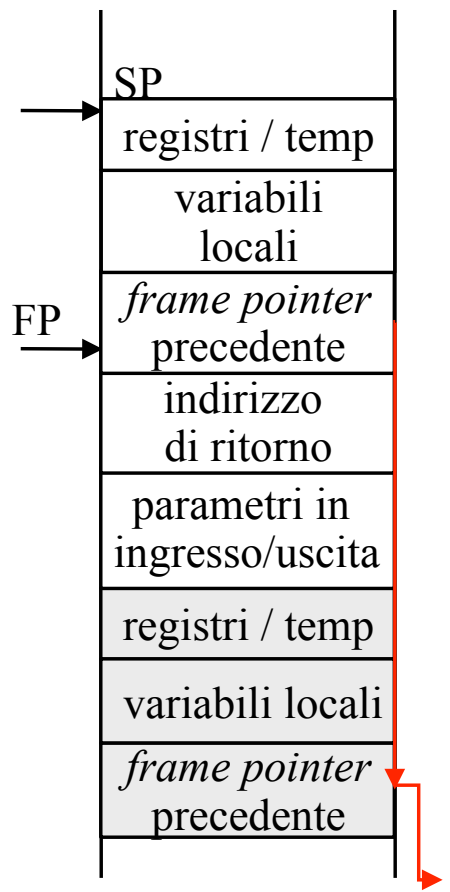
2. [A6] ← [M([SP])]

3. [SP] ← [SP] + 4
- la nuova cima della pila è il FP precedente

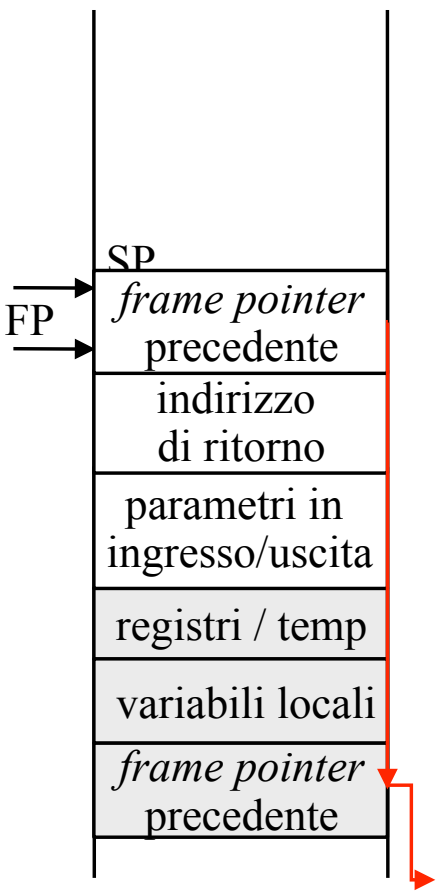
ripristino il vecchio *frame pointer*

abbasso la cima della pila di una parola lunga

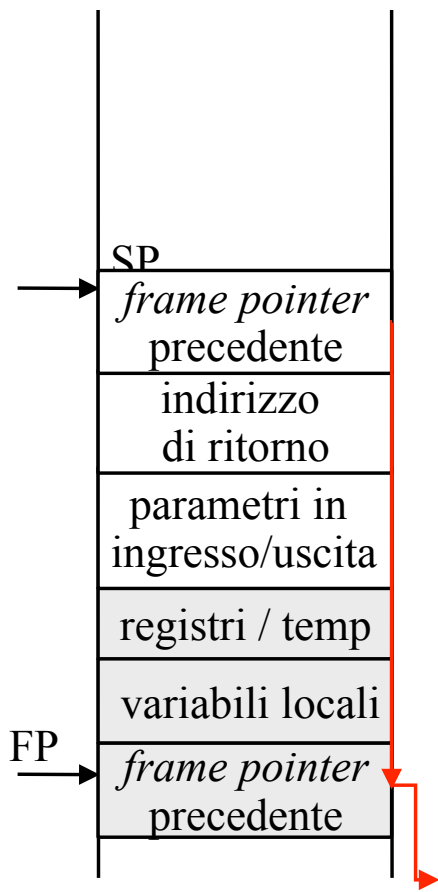
prima del punto 1



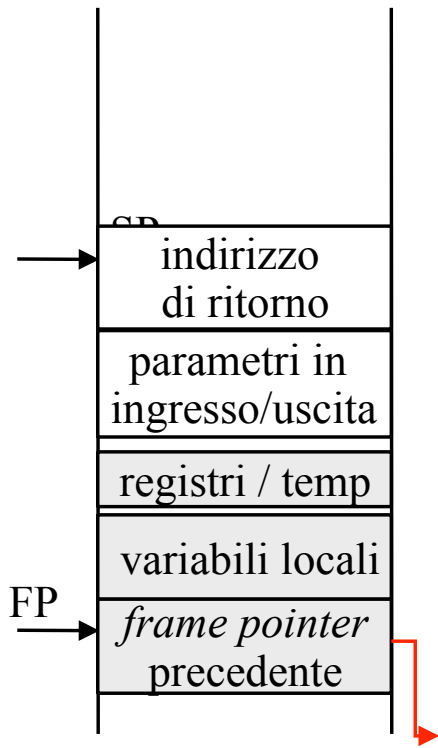
dopo il punto 1



dopo il punto 2



dopo il punto 3
(pronto per RTS)



Traduzione

CONSTATAZIONE

- ❑ la variabile è sempre collocata in memoria
- ❑ gran parte delle istruzioni lavora solo nei registri (o ha vincoli di ortogonalità tali da limitarne l'uso in memoria)

REGOLA BASE PER TRADURRE

- ❑ se hai uno statement C che usa e magari modifica una variabile (per esempio l'assegnamento $a = a + b + 1$), comportati così con la variabile (nell'esempio a)
 - carica nel registro all'inizio dello statement (o non appena serve)
 - e memorizzala (se è stata modificata) alla fine dello statement
- ❑ se la variabile figura nello statement C successivo, non tentare di “tenerla” nel registro – **memorizzala e ricaricala !**

Traduzione

Variabili Globali (scalari e vettori)

intero ordinario e corto a 32 e 16 bit rispettivamente

```
char c;  
int a;  
int b = 5;  
int vet [10];  
int * punt;  
short int d;
```

```
ORG 1000 // decimale  
C: DS.B 1 // spazio per 1 byte  
A: DS.L 1 // oppure DS.B 4  
B: DC.L 5 // inizializzazione  
VET: DS.L 10 // oppure DS.B 40  
PUNT: DS.L 1 // oppure DS.B 4  
D: DS.W 1 // oppure DS.B 2
```

DS riserva solo spazio senza iniziarlo
DC riserva spazio e lo inizializza
il puntatore (di ogni tipo) è equiparato all'intero

Traduzione Variabili Globali - Struct

intero ordinario a 32 bit

```
struct s {  
    char c;  
    int a;  
}
```

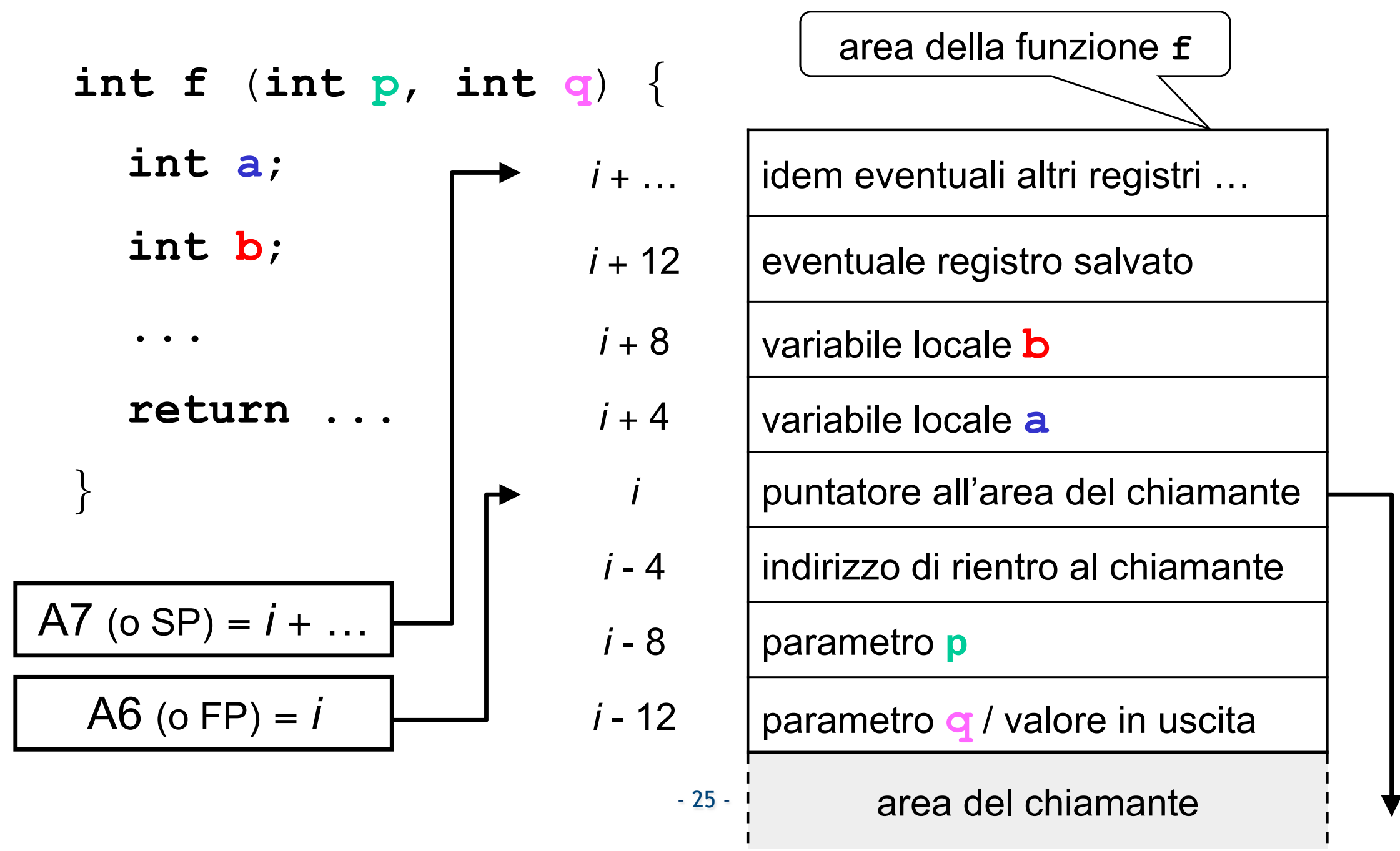
tab. dei sim.	
s	1000
s.c	0
s.A	1

```
ORG 1000 // decimale  
s: DS.B 5 // = somma ingombri di c e a  
s.c: EQU 0 // spiazzamento di c in s  
s.A: EQU 1 // spiazzamento di a in s
```

i campi **c** e **a** ingombrano un byte e una parola lunga, rispettivamente, pertanto la struct **s** ingombra cinque byte; la direttiva DS assegna cinque byte a **s**; la direttiva EQU dichiara gli spiazzamenti **s.c** e **s.A** dei campi **c** e **a**, rispettivamente, all'interno di **s**; il compilatore rinomina gli spiazzamenti, incorporando il nome della struct, per non confonderli con quelli di eventuali campi omonimi in altre struct

Traduzione

Chiamata a funzione - area di attivazione



Traduzione

Chiamata a funzione - variabili locali

```
// chiamante
... // impila parametri
// funzione
int f (int p, int q) {
    int a;      // dich.
    int b;      // dich.
    ...        // elabora
    return ...  // uscita
}
```

tab. dei sim.	
p	-8
Q	-12
A	4
B	8

```
F:      LINK FP, #-8
Q:      EQU   -12 // spi. par. q
P:      EQU   -8  // spi. par. p
A:      EQU    4   // spi. var. a
B:      EQU    8   // spi. var. b
...     // elabora
...     // sovrascrive Q
UNLK FP
RFS
```

EQU dichiara il simbolo senza riservare spazio di memoria; lo spazio viene riservato dal chiamante (impilando i parametri) e da LINK (nella funzione)

ATTENZIONE

**I CONCETTI EVIDENZIATI NEL RIPASSO
NON SONO
SUFFICIENTI PER PASSARE L'ESAME**

Sono solo un piccolo richiamo di alcuni concetti *particolari*
su cui focalizzare la propria attenzione

Esercizio 1

Si vuole **tradurre** in linguaggio macchina simbolico 68000 (linguaggio assembler) il frammento di programma riportato qui sotto: programma principale **main** e funzioni **F** e **G**. Nel tradurre non si tenti di ottimizzare accorpondo istruzioni C indipendenti. La memoria ha indirizzo da **32 bit** ed è indirizzabile per **byte**. Le variabili intere sono da **32 bit**. Si noti che:

- il programma principale **main** è una funzione e ha area di attivazione, ma non restituisce nulla; esso non ha parametri e ha una variabile locale intera; chiama la funzione **F** come procedura, passandole come parametro l'indirizzo di un elemento del vettore *vet*, che è una variabile globale
- la funzione **F** non restituisce nulla; essa ha un parametro puntatore a carattere (ossia a byte ovvero a intero da 8 bit) e non ha variabili locali; chiama la funzione **G**; ha effetto collaterale sulla variabile globale *vet* tramite il parametro puntatore *p*
- la funzione **G** restituisce un valore carattere; essa ha un parametro carattere e non ha variabili locali

Le funzioni salvano i registri utilizzati, e le convenzioni per il passaggio di parametri e valore di uscita sono:

- il parametro, se esiste, viene passato sulla **pila**
- il valore di uscita, se esiste, viene sovrascritto al **parametro** sulla **pila**

```
/* programma principale */
#define N 3      /* costante simbolica */
char vet [N];    /* variabile globale */
void main ( ) {
    int i;        /* variabile locale */
    for (i = 0; i < N; i++) {
        F (&vet [i]);
    } /* for */
    return;
} /* main */
```

```
/* funzione F */
void F (char * p) {
    *p = G (*p);
    return;
} /* F */
```

```
/* funzione G */
char G (char c) {
    return (c * (c - 1) + 2);
} /* G */
```

Si chiede di scrivere quanto segue:

1. in tabella **1.a** le costanti e le variabili globali, in tabella **1.b** l'area di attivazione del programma principale **main** e in tabella **1.c** il codice in linguaggio macchina 68000 di **main**, secondo le convenzioni del corso
2. in tabella **2.a** l'area di attivazione della funzione **G** e in tabella **2.b** il codice in linguaggio macchina 68000 di **G**, secondo le convenzioni del corso; inoltre si disegni l'albero sintattico dell'espressione in **G**
3. in tabella **3.a** l'area di attivazione della funzione **F**, e in tabella **3.b** il contenuto **simbolico** (espresso usando i nomi delle variabili, parametri, registri, ecc), e dove possibile anche i valori **numerici**, e i valori dell'**indirizzo** delle aree di attivazione impilate di **main**, di **F** e di **G**, durante la **seconda** iterazione del ciclo **for** (ossia quando si ha $i = 1$), così come risultano subito dopo l'esecuzione dell'istruzione LINK e l'eventuale salvataggio di registri

Esercizio 2

Si vuole **tradurre** in linguaggio macchina simbolico (linguaggio assembler) 68000 la funzione ricorsiva ***F*** riportata qui sotto. La funzione ***F*** invoca anche la funzione ***G***, di cui è riportata la testata. Nel tradurre non si tenti di accorpate od ottimizzare insieme istruzioni C indipendenti. La memoria ha indirizzo da **32 bit** ed è indirizzabile **per byte**. Le variabili intere sono da **32 bit**.

Le funzioni salvano sempre i registri utilizzati e le convenzioni per il passaggio dei parametri e del valore di ritorno sono le seguenti:

- i parametri sono passati sulla **pila** e caricati in ordine inverso di elencazione (cioè il primo parametro elencato nella testata della funzione è l'ultimo a essere caricato sulla pila)
- il valore di uscita è **sovrascritto** al primo parametro caricato sulla pila

Si chiede di svolgere i punti seguenti:

1. **Si scriva** in tabella 1 il contenuto simbolico (cioè nomi di variabili, di parametri, ecc) della pila subito dopo l'esecuzione dell'istruzione LINK in seguito a un'invocazione di ***F***.
2. **Si scriva** in tabella 2 il codice in linguaggio macchina 68000 di ***F*** tradotta con le convenzioni del corso.

funzione ***F*** in linguaggio C

```

/* testata funzione G */
int G (int);

/* funzione F */
int F (int x, int y) {
    /* variabile locale */
    int i;

    /* parte esecutiva */
    i = x * y;
    if (x <= 0) return G ( y + i );
    else return y * F ( x - y, G ( y ) );
} /* fine F */

```

Esercizio 2

Si supponga che la memoria abbia parole da **16 bit** e sia indirizzata per **byte**. Si svolgano i punti seguenti:

a) Nella tabella sotto, **si riportino** gli indirizzi dove collocare dati e istruzioni. Si consideri che ogni istruzione ha sempre una parola di codice operativo e, se serve, una o più parole aggiuntive. Si tenga conto che lo spiazzamento sia in istruzioni che manipolano dati sia in istruzioni di salto è sempre da **16 bit** e che indirizzi e costanti sono **corti** (16 bit) o **lunghi** (32 bit) come specifica l'istruzione.

		# parole di memoria	# byte	indirizzo (in decimale)
	ORG 1000			
N:	EQU -10			
B:	DS.W 10			
A:	DC.L 10			
	MOVE A.L, D2			
CICLO:	CMP #N.L, D2			
	BLT FINE			
START:	MOVE A.L, D2			
	MOVEA B.L, A0			
	MOVE (A0), D1			
	SUB D1, D2			
	BRA CICLO			
	BSR FUNZ			
FINE:	END START			

Esercizio 3

Si vuole **tradurre** in linguaggio macchina simbolico (linguaggio assembler) 68000 il frammento di programma (funzioni **call** e **f_ric**) riportato qui sotto. Nel tradurre non si tenti di accorpare od ottimizzare insieme istruzioni C indipendenti. La memoria ha indirizzo da **32 bit** ed è indirizzabile **per byte**. Le variabili intere sono da **32 bit**. Si noti che:

- la funzione **call** non restituisce nulla (di fatto è una procedura), non ha parametri e ha due variabili locali intere; essa chiama la funzione **f_ric** (che è una funzione a tutti gli effetti), trattandola come una procedura ossia disinteressandosi del valore in uscita di **f_ric**
- la funzione ricorsiva **f_ric** restituisce un valore intero, ha due parametri e non ha variabili locali

Le funzioni salvano sempre i registri utilizzati e le convenzioni per il passaggio dei parametri e del valore di ritorno sono le seguenti:

- i parametri vengono passati sulla **pila** e caricati in ordine inverso di elencazione (cioè il primo parametro elencato nella testata della funzione è l'ultimo da caricare sulla pila)
- il valore di uscita, se esiste, viene sovrascritto al **primo parametro caricato sulla pila**

Si chiede di scrivere:

- in tabella 1.a e in tabella 2.a il codice in linguaggio macchina 68000 delle funzioni **call** e **f_ric**, rispettivamente, tradotte secondo le convenzioni del corso
- in tabella 1.b e in tabella 2.b il contenuto **simbolico** (espresso usando i nomi delle variabili, parametri, registri, ecc) e i **valori dell'indirizzo** dell'area di attivazione di **call** e di quella relativa alla prima invocazione di **f_ric**, così come risultano **subito dopo l'esecuzione dell'istruzione LINK e il salvataggio di eventuali registri**

In modo coerente con quanto risposto nella domanda precedente, si chiede ora di riportare nella tabella 3 i valori effettivi di parametri e variabili (ove disponibili) presenti nelle aree di attivazione di **call** e di **f_ric** subito dopo il salvataggio dei registri relativo alla **terza invocazione di f_ric**.

```
/* funzione call */
```

```
void call ( ) {  
    int i;  
    int j = 3;  
    i = 0;  
    f_ric (&i, j);  
    return;  
} /* call */
```

```
/* funzione f_ric */
```

```
int f_ric (int *p, int q) {  
    if (q == 0) {  
        return 0;  
    } else {  
        *p = *p + 1;  
        return f_ric (p, q - 1);  
    } /* if */  
} /* f_ric */
```

Esercizio 3

Si vuole **tradurre** in linguaggio macchina simbolico (linguaggio assembler) 68000 il frammento di programma (funzioni **call** e **f_ric**) riportato qui sotto. Nel tradurre non si tenti di accorpare od ottimizzare insieme istruzioni C indipendenti. La memoria ha indirizzo da **32 bit** ed è indirizzabile **per byte**. Le variabili intere sono da **32 bit**. Si noti che:

- la funzione **call** non restituisce nulla (di fatto è una procedura), non ha parametri e ha due variabili locali intere; essa chiama la funzione **f_ric** (che è una funzione a tutti gli effetti), trattandola come una procedura ossia disinteressandosi del valore in uscita di **f_ric**
- la funzione ricorsiva **f_ric** restituisce un valore intero, ha due parametri e non ha variabili locali

Le funzioni salvano sempre i registri utilizzati e le convenzioni per il passaggio dei parametri e del valore di ritorno sono le seguenti:

- i parametri vengono passati sulla **pila** e caricati in ordine inverso di elencazione (cioè il primo parametro elencato nella testata della funzione è l'ultimo da caricare sulla pila)
- il valore di uscita, se esiste, viene sovrascritto al **primo parametro caricato sulla pila**

Si chiede di scrivere:

- in tabella 1.a e in tabella 2.a il codice in linguaggio macchina 68000 delle funzioni **call** e **f_ric**, rispettivamente, tradotte secondo le convenzioni del corso
- in tabella 1.b e in tabella 2.b il contenuto **simbolico** (espresso usando i nomi delle variabili, parametri, registri, ecc) e i **valori dell'indirizzo** dell'area di attivazione di **call** e di quella relativa alla prima invocazione di **f_ric**, così come risultano **subito dopo l'esecuzione dell'istruzione LINK e il salvataggio di eventuali registri**

In modo coerente con quanto risposto nella domanda precedente, si chiede ora di riportare nella tabella 3 i valori effettivi di parametri e variabili (ove disponibili) presenti nelle aree di attivazione di **call** e di **f_ric** subito dopo il salvataggio dei registri relativo alla **terza invocazione di f_ric**.

```
/* funzione call */
```

```
void call ( ) {  
    int i;  
    int j = 3;  
    i = 0;  
    f_ric (&i, j);  
    return;  
} /* call */
```

```
/* funzione f_ric */
```

```
int f_ric (int *p, int q) {  
    if (q == 0) {  
        return 0;  
    } else {  
        *p = *p + 1;  
        return f_ric (p, q - 1);  
    } /* if */  
} /* f_ric */
```

Introduzione al design di architetture hardware su FPGA

Iscrizione richiesta

Giovedì 22 Gennaio 2015

Aula S05

Scaricate la macchina virtuale prima del laboratorio

Il download è di 30GB

Vi servono almeno 80GB sul vostro PC

Consiglio di venire in gruppi di 3-4 persone

www.alessandronacci.it/fpga-acso-2015



That's all Folks!

... in bocca al lupo!
E STUDiate!

Alla prossima lezione

alessandro.nacci@polimi.it