FASCICOLO ASSEMBLER PER RISC-V (istruzioni tratte da ISA RV64i)

I registri *rd*, *rs1* e *rs2* elencati nelle (pseudo-)istruzioni indicano registri a 64 bit da *x0* a *x31*.

CONTROLLO – pseudo-istruzioni

nop			istruzione *nulla*

ARITMETICA – istruzioni native (operazioni in aritmetica in complemento a 2) – vedi nota 1

add	rd, rs1, rs2	rd := rs1 + rs2	*addizione* a 64 bit
addi	rd, rs1, cost12	rd := rs1 + cost12	*addizione* a 64 bit di *costante*
sub	rd, rs1, rs2	rd := rs1 − rs2	*sottrazione* a 64 bit

L'istruzione *oddi* prima estende in segno la costante *cost12* da 12 bit fino a 64 bit, poi la somma al registro sorgente *rs1* e infine scrive il risultato nel registro destinazione *rd*.

ARITMETICA – pseudo-istruzioni (operazioni in aritmetica in complemento a 2)

neg	rd, rs1	rd := −rs1	*negazione* aritmetica (inversione di segno in compl. a 2)

Nota 1: esistono anche le istruzioni macchina native di moltiplicazione (*mul*) e divisione intera, con calcolo separato di quoziente (*div*) e resto (*rem*), in aritmetica in complemento a 2 con operandi e risultato da 64 bit; ecco le istruzioni base: *mul* / *div* / *rem rd*, *rs1*, *rs2*; queste istruzioni calcolano *rd* := *rs1* × *rs2* / *rs1* ÷ *rs2* / *rs1 mod rs2*, rispettivamente; appartengono allo ISA esteso RV64im e hanno numerose varianti.

CONFRONTO – istruzioni native (confronto in aritmetica in complemento a 2)

slt	rd, rs1, rs2	**if** rs1 < rs2 **then** rd := 1 **else** rd := 0	confronto di *minoranza stretta*
slti	rd, rs1, cost12	**if** rs1 < cost12 **then** rd := 1 **else** rd := 0	confronto di *minoranza stretta* rispetto a *costante*

L'istruzione *slti* prima estende in segno la costante *cost12* da 12 bit fino a 64 bit, poi la confronta con il registro sorgente *rs1* e infine scrive il risultato nel registro destinazione *rd*. Le istruzioni native *sltu* e *sltiu* sono definite analogamente, ma il confronto è fatto in aritmetica naturale.

LOGICA – istruzioni native

or	rd, rs1, rs2	rd := rs1 **or** rs2	*somma* logica bit a bit
ori	rd, rs1, cost12	rd := rs1 **or** cost12	*somma* logica bit a bit con *costante* (est. in segno)
and	rd, rs1, rs2	rd := rs1 **and** rs2	*prodotto* logico bit a bit
andi	rd, rs1, cost12	rd := rs1 **and** cost12	*prodotto* logico bit a bit con *costante* (idem)
xor	rd, rs1, rs2	rd := rs1 x**or** rs2	*somma* logica *esclusiva* bit a bit
xori	rd, rs1, cost12	rd := rs1 x**or** cost12	*somma* logica *esclusiva* bit a bit con *cost.* (idem)

Le istruzioni *ori*, *andi* e *xori* prima estendono in segno la costante *cost12* da 12 bit fino a 64 bit, poi la compongono bit a bit con il registro sorgente *rs1* e infine scrivono il risultato nel registro destinazione *rd*.

LOGICA – pseudo-istruzioni

not	rd, rs1	rd := **not** rs1	*negazione* logica bit a bit (complemento bit a bit)

SCORRIMENTO – istruzioni native (costante intera positiva compresa nell'intervallo da 1 a 63)

slli	rd, rs1, cost12	rd := rs1 << cost12	scorrimento *logico verso sx* del numero di posizioni specificato da costante (introduce bit 0 da *dx*)
srli	rd, rs1, cost12	rd := rs1 >> cost12	scorrimento *logico verso dx* del numero di posizioni specificato da costante (introduce bit 0 da *sx*)
srai	rd, rs1, cost12	rd := rs1 >> cost12	scorrimento *aritmetico verso dx* del numero di posizioni specificato da costante (estende il segno)

Il terzo argomento, che dà il numero di posizioni da fare scorrere, può essere contenuto in un registro. Per esempio, l'istruzione nativa *sll rd, rs1, rs2* lo specifica nel registro *rs2*; idem per le istruzioni native *srl* e *sra*.

SALTO INCONDIZIONATO – istruzioni native – vedi nota 2

jal	rd, spi20	rd := pc + 4 pc := pc + spi20	salto *incondizionato* relativo a PC con *salvataggio* dell'indirizzo di rientro nel registro *rd*
jalr	rd, spi12(rs1)	rd := pc + 4 pc := rs1 + spi12	salto *incondizionato* relativo a registro *rs1* con *salvataggio* dell'indirizzo di rientro nel registro *rd*

SALTO INCONDIZIONATO – pseudo-istruzioni – vedi nota 2

j	spi20	pc := pc + spi20	*salto incondizionato* – salto relativo a PC
jr	rs1	pc := rs1	*salto incondizionato a lunga distanza* – salto da registro *rs1*
jal	spi20	ra := pc + 4 pc := pc + spi20	*chiamata a sottoprogramma* – salto incondizionato relativo a PC con salvataggio dell'indirizzo di rientro nel registro *ra*
jalr	rs1	ra := pc + 4 pc := rs1	*chiamata a sottoprogramma a lunga distanza* – salto incondizionato da registro *rs1* con salvataggio dell'indirizzo di rientro nel registro *ra*
ret		pc := ra	*rientro da sottoprogramma* – salto da registro *ra*

SALTO CONDIZIONATO – istruzioni native (confronto in aritmetica in compl. a 2) – vedi nota

beq	rs1, rs2, spi12	**if** rs1 = rs2 pc := pc + spi13	salta (relativamente a PC) se *uguale*
bne	rs1, rs2, spi12	**if** rs1 ≠ rs2 pc := pc + spi13	salta (idem) se *diverso*
bge	rs1, rs2, spi12	**if** rs1 ≥ rs2 pc := pc + spi13	salta (idem) se *maggiore o uguale*
blt	rs1, rs2, spi12	**if** rs1 < rs2 pc := pc + spi13	salta (idem) se *strettamente minore*

Le istruzioni native *bgeu* e *bltu* sono definite analogamente, ma il confronto è fatto in aritmetica naturale.

SALTO CONDIZIONATO – pseudo-istruzioni (confronto in aritmetica in compl. a 2) – vedi nota 2

bgt	rs1, rs2, spi12	**if** rs1 > rs2 pc := pc + spi13	salta (idem) se *strettamente maggiore*
ble	rs1, rs2, spi12	**if** rs1 ≤ rs2 pc := pc + spi13	salta (idem) se *minore o uguale*

Le pseudo-istruzioni *bgtu* e *bleu* sono definite analogamente, ma il confronto è fatto in aritmetica naturale.

Nota 2: Lo spiazzamento *spi12* o *spi20* (numero in complemento a 2) esprime la distanza di salto (positiva o negativa) in termini di mezze parole (half). L'indirizzo destinazione di salto, che verrà inviato alla memoria durante l'esecuzione del salto per prelevare l'istruzione collocata alla destinazione del salto, fa riferimento al byte. Durante l'esecuzione il processore moltiplica per 2 lo spiazzamento e così lo riferisce a byte. L'istruzione *jalr* (che è in formato I) fa eccezione: il processore *non* moltiplica per 2 lo spiazzamento *spi12*, ma si limita a *sovrascrivere* con 0 il bit meno significativo dello spiazzamento (se era 1 diventa 0, o resta 0).

TRASFERIMENTO DA / A MEMORIA – istruzioni native

ld	rd, spi12 (rs1)	rd := mem [rs1 + spi12]	carica *parola doppia* (da 64 bit)
sd	rs2, spi12 (rs1)	mem [rs1 + spi12] := rs2	memorizza *parola doppia* (da 64 bit)
lw, *lwu*	rd, spi12 (rs1)	rd := mem [rs1 + spi12]	carica *parola* (da 32 bit)
sw	rs2, spi12 (rs1)	mem [rs1 + spi12] := rs2	memorizza *parola* (da 32 bit)
lh, *lhu*	rd, spi12 (rs1)	rd := mem [rs1 + spi12]	carica *mezza parola* (da 16 bit)
sh	rs2, spi12 (rs1)	mem [rs1 + spi12] := rs2	memorizza *mezza parola* (da 16 bit)
lb, *lbu*	rd, spi12 (rs1)	rd := mem [rs1 + spi12]	carica *byte* (da 8 bit)
sb	rs2, spi12 (rs1)	mem [rs1 + spi12] := rs2	memorizza *byte* (da 8 bit)

Le istruzioni di trasferimento tra registri e memoria vanno usate con indirizzamento di memoria allineato, secondo la rispettiva dimensione di parola trasferita (*d*, *w*, *h*, *b*). Lo spiazzamento (in compl. a 2) è espresso come numero di byte. Solitamente esso è specificato come numero o come costante simbolica dichiarata tramite la direttiva *.eqv*. La scrittura *ld rd, (rs1)* sottintende spiazzamento nullo (similmente per tutte le altre *load* e tutte le *store*). Le istruzioni *lw*, *lh* e *lb* estendono in segno la parte superiore (non interessata dal caricamento) del registro destinazione *rd*, mentre le istruzioni *lwu*, *lhu* e *lbu* la azzerano.

TRASFERIMENTO TRA REGISTRI – pseudo-istruzioni

mv	rd, rs1	rd := rs1	*copia* registro

CARICAMENTO DI COSTANTE E INDIRIZZO IN REGISTRO – istruzioni native

lui	rd, cost20	rd := cost20 (solo alcuni bit)	*caricamento (parziale) di costante* – carica la costante *cost20* nei 20 bit (5 cifre hex) più significativi della metà inferiore da 32 bit (8 cifre hex) del registro *rd*, azzera i 12 bit (3 cifre hex) meno significativi di *rd* ed estende in segno la metà inferiore di *rd* fino al 64-esimo bit
auipc	rd, spi20	rd := pc + spi20 (solo alcuni bit)	*caricamento (parziale) di indirizzo relativo a PC* – prima somma lo spiazzamento *spi20* all'indirizzo dell'istruzione *auipc* (contenuto nel registro *pc*), poi lo tratta come la costante nell'istruzione *lui* e lo carica nel registro *rd*

Solitamente la costante *cost20* è specificata come valore o come costante simbolica dichiarata tramite la direttiva *.eqv*. Lo spiazzamento *spi20* esprime la distanza (in byte) e di solito è specificato come etichetta di istruzione o di dato. Le istruzioni *lui* e *auipc* sono usate per espandere le pseudo-istruzioni *li* e *la*.

CARICAMENTO DI COSTANTE E INDIRIZZO IN REGISTRO – pseudo-istruzioni

li	rd, cost32	rd := cost32	*caricamento (completo) di costante*
la	rd, spi32	rd := pc + spi32	*caricamento (completo) di indirizzo relativo a PC*

Entrambe le pseudo-istruzioni *li* e *la* prima caricano un numero da 32 bit (word – 8 cifre hex), *cost32* e *spi32* rispettivamente, nella metà inferiore del registro destinazione *rd*, e poi estendono il segno della metà inferiore di *rd* alla metà superiore di *rd*. La pseudo-istruzione *li (load immediate*) carica una costante, di solito specificata come valore o come costante simbolica dichiarata tramite direttiva *.eqv*. La pseudo-istruzione *la* (*load address*) carica un indirizzo relativo a PC, di solito specificato come etichetta di istruzione macchina o di direttiva di dichiarazione di dato. Per vedere le espansioni di *li* e *la*, e come viene trattato e suddiviso l'eventuale simbolo che esprime l'argomento *cost32* e *spi32*, si legga di seguito.

COME VENGONO ESPANSE LE PSEUDO-ISTRUZIONI *LI* e *LA*

L'argomento della pseudo-istruzione *li* / *la* può essere un valore / un indirizzo da 32 bit già in forma numerica, oppure una costante simbolica / un'etichetta di istruzione o di dato. Se esso è una costante simbolica / un'etichetta, viene risolto nel corrispondente valore / indirizzo, e diventa un numero da 32 bit:

pseudo-istruzione	**risoluzione dell'argomento**
li rd, costante simbolica o valore	**li** rd, valore // num da 32 bit
la rd, etichetta o indirizzo	**la** rd, indirizzo // num da 32 bit

Inoltre per la pseudo-istruzione *la* (ma non per *li*) l'indirizzo viene reso relativo a PC, cioè viene convertito in distanza (*delta*) rispetto al registro *program counter* corrente (il quale contiene l'indirizzo dove è collocata *la*). La pseudo-istruzione *li* / *la* viene espansa nelle istruzioni native *lui* / *auipc* e (per entrambe) *addi*, suddividendo il numero da 32 bit in due componenti: 20 bit superiori (*hi*) e 12 bit inferiori (*lo*). Per rendere tale numero relativo a PC (*pcrel*) e suddividerlo, l'assemblatore utilizza questi modificatori:

modificatore	**calcolo del modificatore** - da parte dal collegatore
%**hi**(val)	= val[31:12] + val[11] // 20 bit sup corretti
%**lo**(val)	= val[11:0] // 12 bit inf
si ponga: delta = ind – pc // l'indirizzo viene relativizzato a PC	
%**pcrel_hi**(ind)	= delta[31:12] + delta[11] // 20 bit sup corretti
%**pcrel_lo**(ind)	= delta[11:0] // 12 bit inf

Qui *val* / *ind* indica un numero da 32 bit, cioè un valore / un indirizzo dato come numero oppure ricavato risolvendo una costante simbolica / un'etichetta di istruzione o dato, e *delta* indica una distanza di salto relativamente a PC. I bit di *val* / *delta* sono numerati da 0 (bit meno significativo) a 31 (bit più significativo). L'assemblatore applica automaticamente i modificatori espandendo le pseudo-istruzioni *li* / *la*.

Ecco l'espansione di *li* / *la* tramite le istruzioni native e i modificatori predetti, e sotto la motivazione:

pseudo-istruzione	**espansione** - in istruzioni native e modificatori
li rd, valore	**lui** rd, %**hi**(valore) // 20 bit sup - HI **addi** rd, rd, %**lo**(valore) // 12 bit inf - LO
la rd, indirizzo	**auipc** rd, %**pcrel_hi**(indir) // 20 bit sup - HI **addi** rd, rd, %**pcrel_lo**(indir) // 12 bit inf - LO

L'istruzione *addi* prima estende in segno la costante da 12 bit fino a 64 bit e poi la somma al registro *rd*. Se il bit più significativo della costante vale 1, cioè se si ha *valore* / *delta* [*11*] = 1, dopo l'estensione la costante avrà assunto valore negativo. Poiché la costante esprime il componente da 12 bit, che in origine è positivo o nullo, occorre correggerla. Sommando il bit *valore* / *delta* [*11*] al componente da 20 bit, si corregge l'effetto dell'estensione di segno. L'assemblatore, generando il modulo oggetto, espande la pseudo-istruzione *li* / *la* tramite i modificatori. Il collegatore, unendo i moduli oggetto e generando l'eseguibile, calcola i modificatori e produce i componenti corretti da 20 bit e 12 bit che figurano in *lui* / *auipc* e *addi*.

COME VENGONO ESPANSE LE PSEUDO-ISTRUZIONI *NOP NEG NOT MV* e *BGT BLE*

pseudo-istruzione	**espansione** - in istruzione nativa
nop	**addi** zero, zero, 0 // reg zero è immodificabile
neg rd, rs1	**sub** rd, zero, rs1 // rd = zero – rs1 = -rs1
not rd, rs1	**xori** rd, rs1, -1 // rd = rs1 xor 1^{64} = not rs1
mv rd, rs1	**addi** rd, rs1, 0 // rd = rs1 + 0 = rs1
bgt rs1, rs2, spi	**blt** rs2, rs1, spi // scambia rs1>rs2 con rs2<rs1
ble rs1, rs2, spi	**bge** rs2, rs1, spi // scambia rs1<=rs2 con rs2>=rs1

COME VENGONO ESPANSE LE PSEUDO-ISTRUZIONI DI SALTO INCOND. *J JR JAL JALR* e *RET*

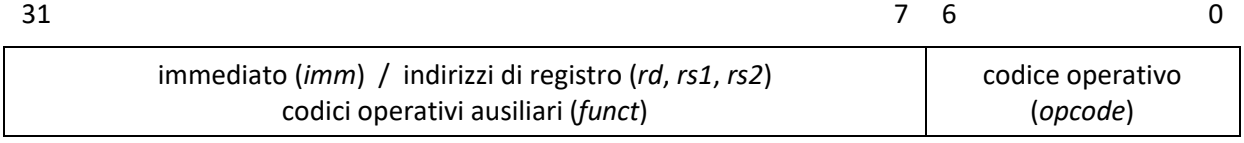
pseudo-istruzione	**espansione** - in istr. nativa **jal**(**r**) rd, spi(rs1)
j spi	**jal** zero, spi // non salva ind. di rientro
jr rs1	**jalr** zero, 0(rs1) // non salva ind. di rientro
jal spi	**jal** ra, spi // salva in ra ind. di rientro
jalr rs1	**jalr** ra, 0(rs1) // salva in ra ind. di rientro
ret	**jalr** zero, 0(ra) // prende da ra ind. di rientro

L'istruzione nativa *jal* effettua un salto con spiazzamento *spi* e salva l'indirizzo di rientro ($pc + 4$) nel registro destinazione *rd*. L'istruzione nativa *jalr* opera similmente, ma salta a registro base (registro sorgente) *rs1* più spiazzamento *spi*. Rimappando come sopra gli argomenti di queste due istruzioni native si ottengono tutte le pseudo-istruzioni elencate (*rd* = *zero* ha come effetto quello di non salvare l'indirizzo di rientro).

COME INDICARE I COMPONENTI DI UN'ISTRUZIONE MACCHINA E DI UN IMMEDIATO

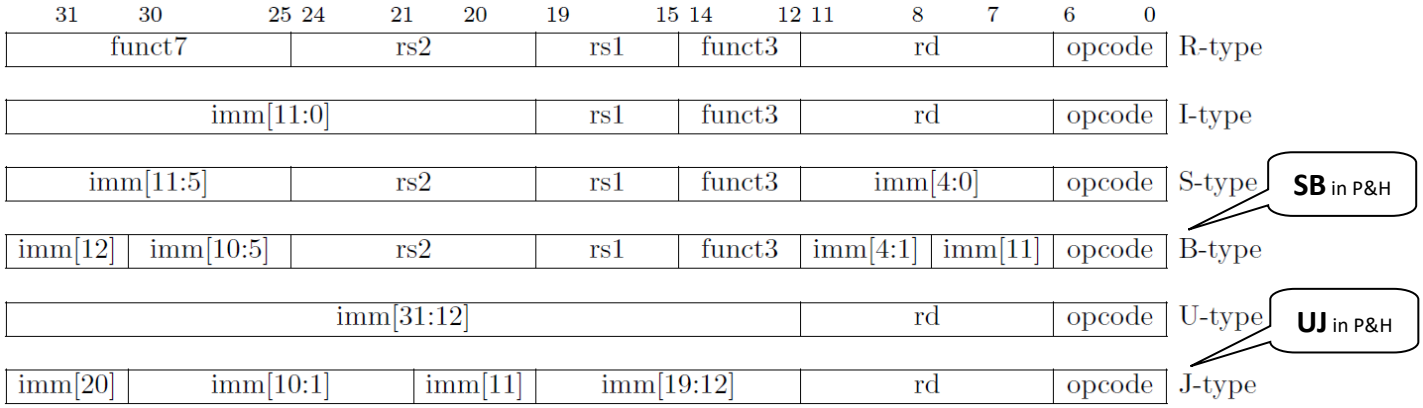
Le istruzioni macchina di ISA RV64i sono codificate come parole da 32 bit. I bit dell'istruzione *inst* sono numerati da 0 a 31 (inclusi) e si indicano individualmente con *inst*[*k*] (con $31 \geq k \geq 0$). La notazione *inst*[*k*:*h*] (con $31 \geq k > h \geq 0$) indica il campo di $k - h + 1 \geq 2$ bit compreso tra i bit k esimo e h esimo (inclusi) di *inst*. Similmente un immediato *imm* (costante o spiazzamento) ha varia lunghezza (secondo il *formato* dell'istruzione cui appartiene) e i suoi bit e campi di bit si indicano con *imm*[*k*] e *imm*[*k*:*h*], rispettivamente.

istruzione macchina nativa *inst* (ISA RV64i) – i campi di bit da 31 a 7 (inclusi) dipendono dal *formato*



Il campo *inst*[*6:0*] è di 7 bit e codifica sempre il *codice operativo* dell'istruzione *inst*. I bit rimanenti [*31:7*] sono suddivisi in vari campi e codificano un eventuale *immediato*, gli indirizzi dei *registri* dati come argomenti, ed eventuali *codici operativi ausiliari* (*funct7* e *funct3*), secondo il *formato* dell'istruzione *inst*.

FORMATI NUMERICI A 32 BIT DELLE ISTRUZIONI MACCHINA NATIVE – vedi nota 3

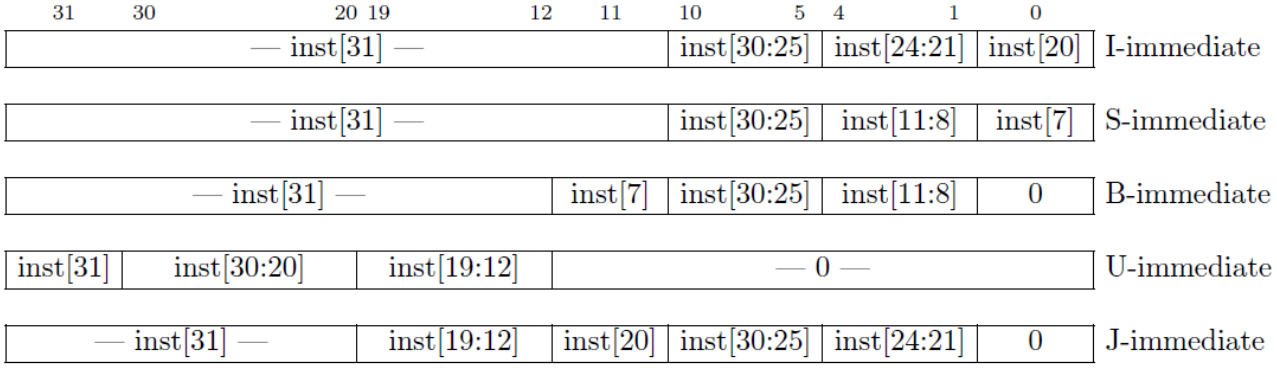


Nel libro di **Patterson & Hennessy** i formati *B* e *J* sono chiamati ***SB*** e ***UJ***, per la loro somiglianza con *S* e *U*.

Nota 3: Il formato *B* (con due registri sorgente) contiene uno spiazzamento *imm*[*12:1*] da 12 bit, che esprime la distanza di salto (positiva o negativa) in termini di mezze parole (*half*). Il processore *moltiplica* lo spiazzamento per 2 (*aggiungendogli* un bit 0 in coda), così *riferendolo a byte*. Questa è l'unica differenza tra i formati *S* e *B*. Similmente per i formati *U* e *J*, con spiazzamento da 20 bit (con un registro destinazione).

COSTRUIRE UN IMMEDIATO DAI BIT DELL'ISTRUZIONE ED ESTENDERLO IN SEGNO FINO A 32 BIT

Durante l'esecuzione, il processore RISC-V costruisce l'immediato (costante o spiazzamento) a 32 bit a partire dai bit *imm* contenuti nell'istruzione *inst*, secondo questi schemi dipendenti dal formato di *inst*.



L'estensione in segno a 32 bit di un immediato (laddove è prevista) viene ottenuta replicando il bit *inst*[*31*] dell'istruzione e qui è indicata con – *inst*[*31*] –. La notazione – 0 – indica una sequenza di bit 0 *aggiunti* in coda. Dopo essere stato costruito a 32 bit, l'immediato può essere ancora esteso in segno fino a 64 bit.

CORRISPONDENZA TRA FORMATI NUMERICI E ISTRUZIONI MACCHINA NATIVE – vedi nota 4

formato	***istruzione***	***aggiunte e varianti***
R	*add, sub, or, and, xor, slt*	*mul, div, rem, sll, srl, sra, sltu*
I	*addi, slti, ori, andi, xori, slli, srli, srai, ld, lw, lh, lb, jalr*	*sltiu, lwu, lhu, lbu*
S	*sd, sw, sh, sb*	
B (o *SB*)	*beq, bne, bge, blt*	*bgeu, bltu*
U	*lui, auipc*	
J (o *UJ*)	*jal*	

Nota 4: L'istruzione *jalr*, pur effettuando un salto a istruzione, ha formato *I* (non *J*) e il processore non ne allinea lo spiazzamento *spi12* a mezza parola (half) aggiungendo in coda il bit 0, come fa per le istruzioni in formato *B* e *J* (nota 2). Invece, per *jalr* il processore *azzera* il bit meno significativo *imm*[0] dello spiazzamento, benché esso figuri nel formato *I* e quivi possa valere 1. Dunque per *jalr* i bit liberi di *spi12* sono solo 11 e la max distanza di salto è la metà di quella per i salti condizionati *bxx* e per l'istruzione *jal*.

REGISTRI REFERENZIABILI – nominabili come argomento di istruzione macchina

#	*nome ABI*	*utilizzo*
0	*zero*	*costante* 0 – immodificabile (anche se viene scritta non cambia valore)
1	*ra*	*return address* – indirizzo di rientro da funzione
2	*sp*	*stack pointer* – puntatore alla cima della pila
3	*gp*	*global pointer* – puntatore all'area dati globale **QUI NON USATO**
4	*tp*	*thread pointer* – puntatore a thread **QUI NON USATO**
5 – 7	*t0 – t2*	*valori temporanei* (indirizzi, variabili globali e calcolo delle espressioni)
8	*s0 / fp*	*variabile locale* di funzione / *frame pointer* (se in uso)
9	*s1*	*variabile locale* di funzione
10 – 11	*a0 – a1*	*valore restituito* da funzione (di solito basta *a0* – se serve si aggiunge *a1*)
12 – 17	*a2 – a7*	*argomenti* in *ingresso* a funzione (max 6 argomenti, gli eccedenti in pila)
18 – 27	*s2 – s11*	*variabili locali* di funzione (max 10 o 11 variabili, le eccedenti in pila)
28 – 31	*t3 – t6*	*valori temporanei* (indirizzi, variabili globali e calcolo delle espressioni)

Nelle (pseudo-)istruzioni macchina i registri vanno indicati di preferenza tramite i loro nomi ABI, perché il nome ABI esprime l'uso del registro. I due usi previsti per il registro *x8* (*s0* oppure *fp*) sono mutuamente esclusivi. Se il frame pointer *fp* non è in uso, il registro *s0* è disponibile per allocare una variabile locale di funzione, altrimenti per allocare si inizia dal registro *s1*. I registri *gp* e *tp* sono previsti per compatibilità con altri modelli di processore (p. es. MIPS) e per programmazione concorrente, ma qui non sono usati.

SALVATAGGIO E RIPRISTINO DEI REGISTRI – salvare / ripristinare i registri in / da pila a 64 bit

ruolo	*registro/i*	*dettagli*
caller-saved	*t0 – t6, a0 – a7*	salvare in / ripristinare da pila (se serve) nell'ordine indicato
callee-saved	*fp, ra, s0 – s11*	salvare in / ripristinare da pila (se serve) nell'ordine indicato

Il registro frame pointer *fp* va salvato e ripristinato solo se è in uso (nel qual caso il registro *s0* non esiste).

DIRETTIVE FONDAMENTALI DI ASSEMBLATORE RISC-V – per RARS e GCC (con eccezioni)

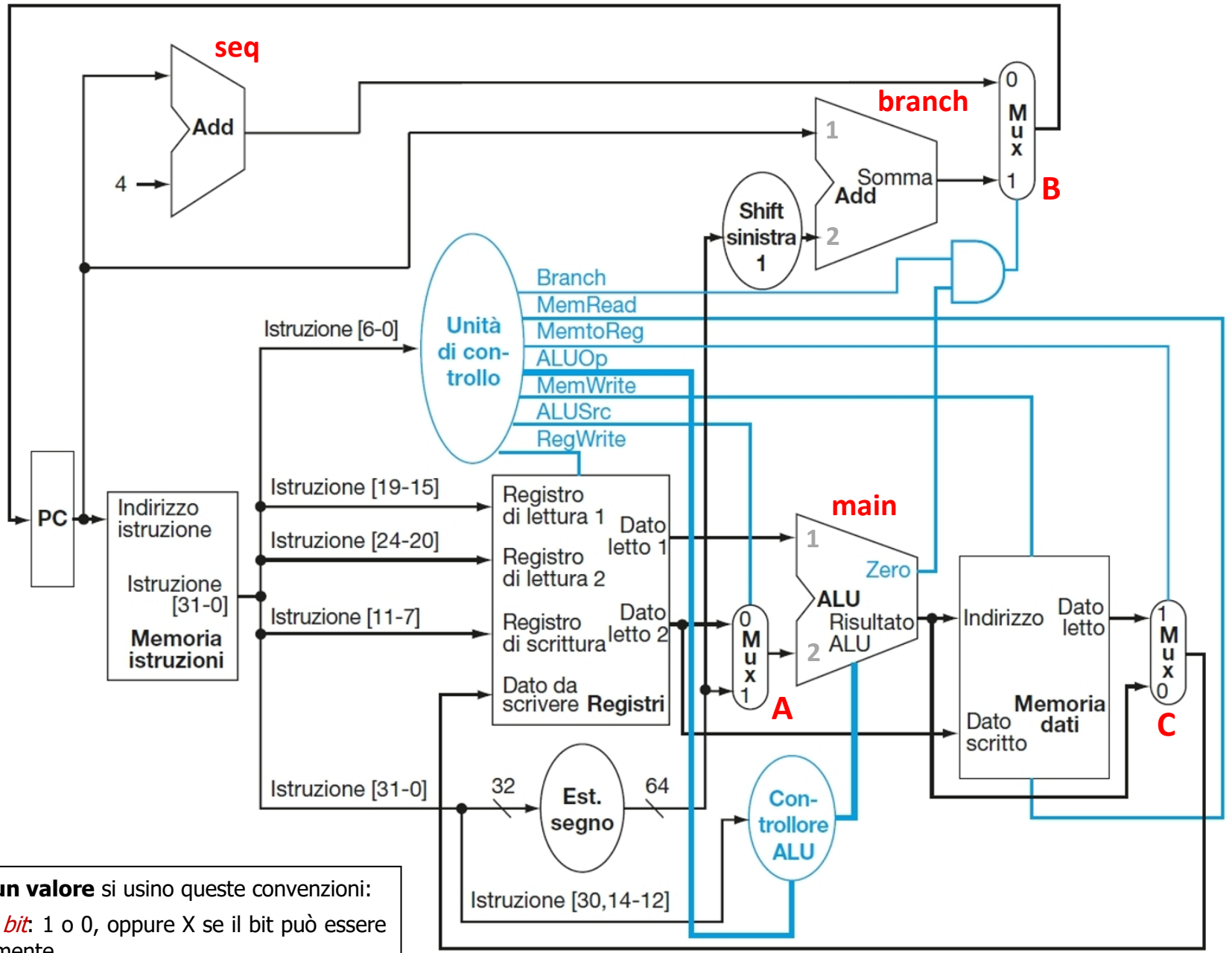
etichetta	*direttiva*	*argomento/i*	*funzione della direttiva*
no	***.align***	*0* o *1* o *2* o *3*	allinea il dato successivo a *byte* o *half* o *word* o *dword*
sì	***.ascii(z)***	"*stringa*"	metti costante *stringa* in memoria (con NULL in fondo)
sì	***.byte***	*c1*, *c2*, ...	metti il/i byte con valore iniziale *c1*, *c2*, ... in memoria
no	***.data***	*ind* a 64 bit	dichiara segmento dati a *ind* (*ind* è opzionale)
sì	***.dword***	*d1*, *d2*, ...	metti parola/e doppia/e (double) con v.i. *d1*, *d2*, ... in mem
no	***.eqv***	*sym*, *val*	dichiara costante simbolica *sym* con valore *val* (#define)
no	***.globl***	*sym*	esporta simbolo *sym* ad altri moduli oggetto
sì	***.half***	*h1*, *h2*, ...	metti mezza/e parola/e (half) con v.i. *h1*, *h2*, ... in mem
sì	***.space***	*num*	alloca *num* byte di spazio non inizializzato – RARS
no	***.text***	*ind* a 64 bit	dichiara segmento testo (codice) a *ind* (*ind* è opzionale)
sì	***.word***	*w1*, *w2*, ...	metti parola/e (word) con v.i. *w1*, *w2*, ... in memoria
sì	***.zero***	*num*	alloca *num* byte di spazio BSS (inizializzato a 0) – GCC

L'allineamento di default è ***.align*** 0, cioè allineamento a byte ossia nessun allineamento: in questo caso le direttive di dichiarazione di dato ***.ascii***(***z***), ***.byte*** ***.dword***, ***.half***, ***.space***, ***.word*** e ***.zero*** allocano lo spazio di memoria richiesto a partire dal primo byte libero. Si suppone che il processore e la memoria, a livello HW, supportino una tale modalità di accesso non allineata (processori/memoria diversi possono differire).

Le direttive ***.space*** e ***.zero*** sono molto simili. La seconda (riconosciuta da GCC) alloca spazio di memoria BSS (Linux), che il sistema operativo Linux inizializza a 0 quando carica il processo in memoria.

Altre direttive (qui non usate): ***.macro*** ed ***.end_macro*** per dichiarare una macro (pseudo-istruzione), e ***.rodata*** (GCC) per dichiarare un segmento di dati non modificabili (*read-only data*). La direttiva ***.rodata*** serve principalmente per dichiarare tabelle globali fisse di costanti lunghe o di puntatori / spiazzamenti (*GOT – Global Offset Table*) a variabili a lunga distanza, o per strutture dati complesse immodificabili.

PROCESSORE SINGOLO CICLO (monociclo)

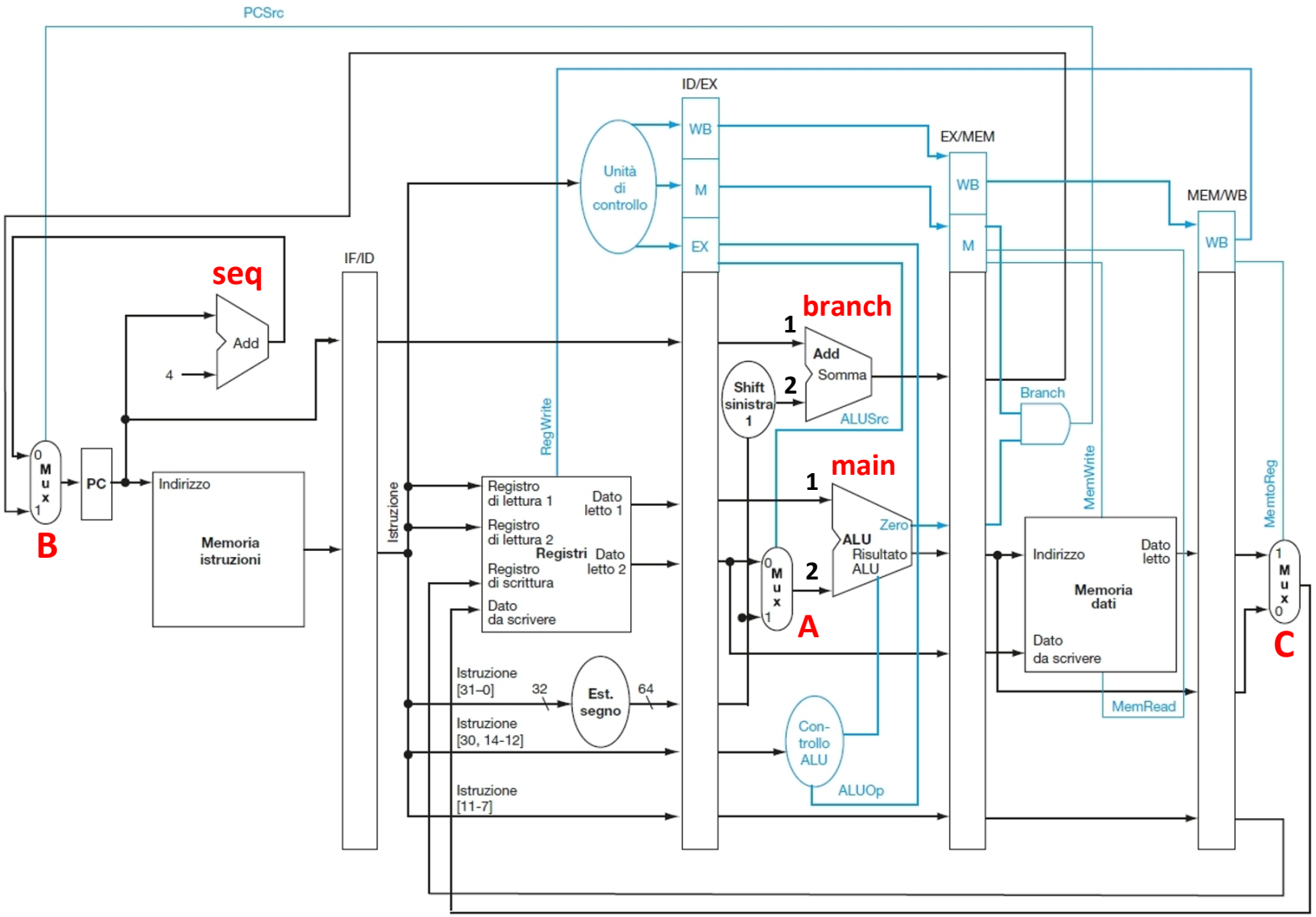


Per **rappresentare un valore** si usino queste convenzioni:

- *segnale a singolo bit*: 1 o 0, oppure X se il bit può essere 1 o 0 indifferentemente
- *valore a più bit*: codifica hex del valore, oppure "n.d." se il valore non è determinabile per il caso richiesto

in grassetto i nomi dei multiplexer **A**, **B** e **C**, e delle tre ALU **main**, **branch** e **seq**

PROCESSORE PIPELINE (multiciclo) e FORMATI DEI REGISTRI INTER-STADIO



Per **rappresentare un valore** si usino queste convenzioni:

- *segnale a singolo bit*: 1 o 0, oppure X se il bit può essere 1 o 0 indifferentemente
- *valore a più bit*: codifica hex del valore, oppure "n.d." se il valore non è determinabile per il caso richiesto

Legenda dei registri inter-stadio

(64) ecc. = numero di bit del campo considerato

(*rs1*) o (*rs2*) = contenuto del registro sorgente *rs1* o *rs2*

rd = numero del registro destinazione

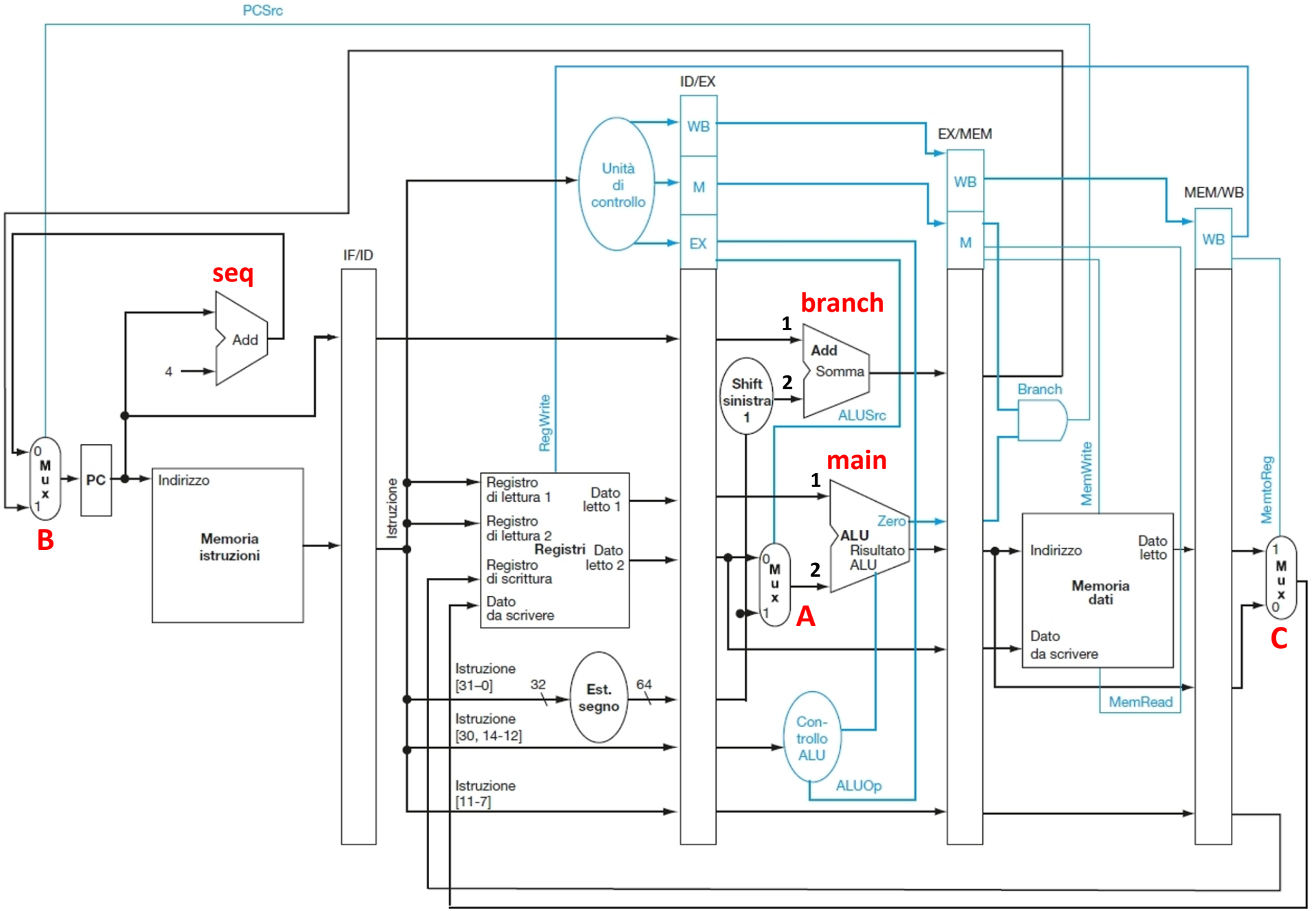
I campi dei registri inter-stadio sono indicati come:

Nome_Registro_Interstadio.Campo_x
(p. es. MEM/WB.dato letto)

IF/ID	PC (64)	istruzione (32)							
ID/EX	WB (2)	M (3)	EX (3)	PC (64)	(*rs1*) (64)	(*rs2*) (64)	offset esteso (64)	funz7_3 (4)	*rd* (5)
EX/MEM	WB (2)	M (3)	PC (64)	ALU_out (64)	bit Z (1)	(*rs2*) (64)	*rd* (5)		
MEM/WB	WB (2)	dato letto (64)		ALU_out (64)	*rd* (5)				

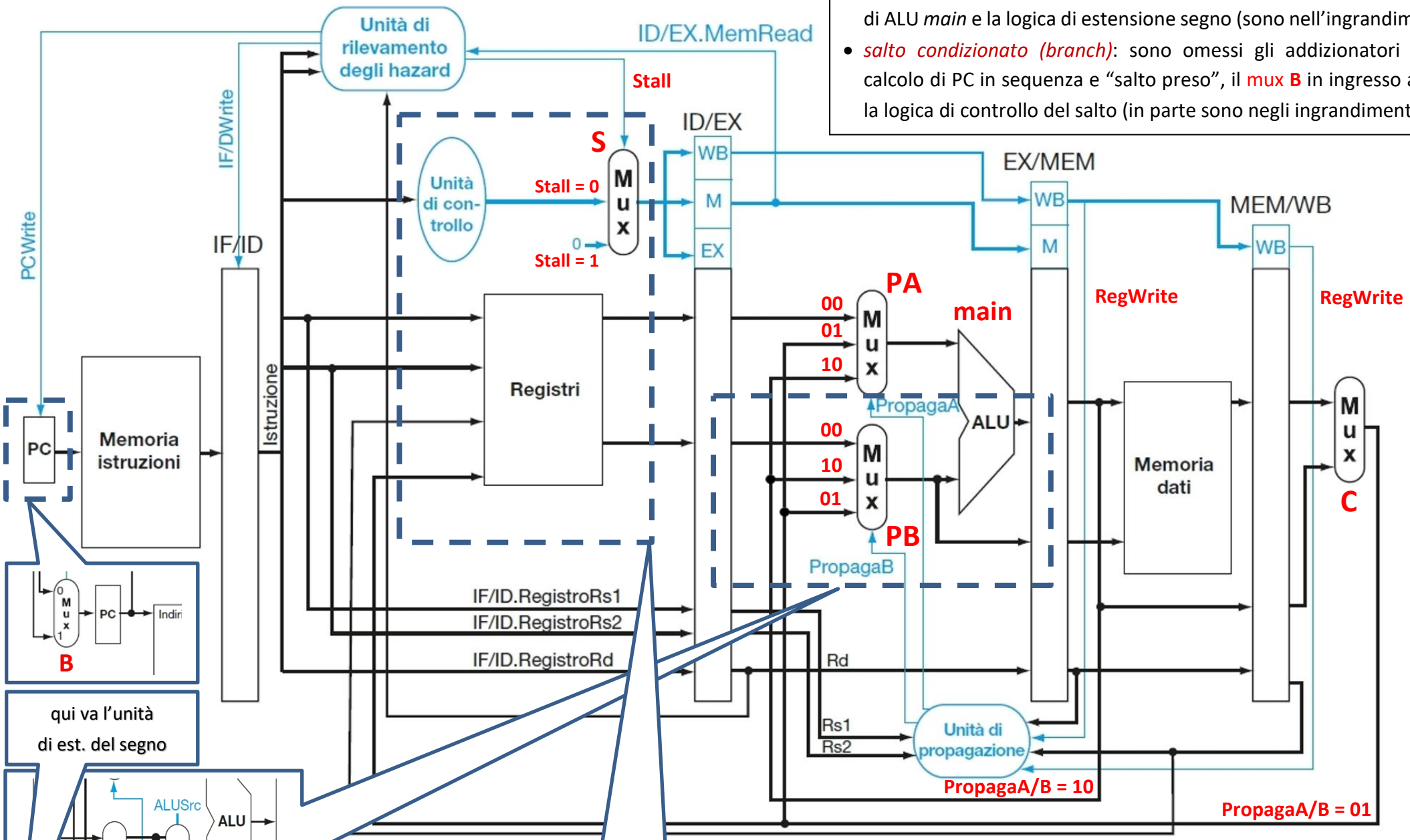
formati dei registri inter-stadio

PROCESSORE PIPELINE (multiciclo) ingrandito per comodità di lettura



UNITÀ di STALLO e UNITÀ di PROPAGAZIONE

- *propagazione*: sono presenti i tre mux **S**, **PA** e **PB**
- *immediato (costante)*: sono omessi il mux **A** al secondo ingresso di ALU *main* e la logica di estensione segno (sono nell'ingrandimento)
- *salto condizionato (branch)*: sono omessi gli addizionatori per il calcolo di PC in sequenza e "salto preso", il mux **B** in ingresso a PC e la logica di controllo del salto (in parte sono negli ingrandimenti)



- *ottimizzazione del salto condizionato (branch)*: confronto dei registri in stadio ID e addizionatore per il calcolo di PC in "salto preso"
- *ingrandimento ruotato; si vedano anche le figure precedenti*