Avevamo visto l’istruzione di tipo BRANCH, che ha le due varianti (a seconda del valore del bit L diventa BRANCH o BRANCH and LINK). L’idea è di andare a sommare un valore costante al valore corrente del PC per determinare una variazione del flusso di esecuzione del codice. Alcuni dettagli: nell’offset abbiamo 24 bit, ma il registro R15 contiene indirizzi espressi su 26 bit, eccetto che gli ultimi due assumono sempre il valore 0 quindi non c’è problema (basta scorrere il valore di offset di due posizioni). L’offset inoltre è un numero con segno in complemento a due.   
I salti sono di tipo relativo (si riferiscono alla posizione corrente). A complicare le cose un pochino c’è la pipeline: essa fa sì che nel momento in cui viene eseguita un’istruzione (essendo una pipeline a 3 stadi – fetch, decodifica e esecuzione) il PC è già stato incrementato due volte, quindi bisogna ricordarsi quando si scrive il codice in assembler che il valore da sommare presente in offset è da sommare non all’indirizzo dell’istruzione, ma all’indirizzo dell’istruzione +8 (due volte +4). Fortunatamente sono i compilatori e gli assemblatori ad occuparsi di questo, ma se noi volessimo scrivere il codice operativo della nostra istruzione dovremmo tenere conto di questo incremento di 8 indirizzi.

Abbiamo saltato le istruzioni CODTRANS, COREGOP, CORTRANS: esse sono tipiche della versione 2 di ARM ma non sono state implementate nel processore Amber. Il CO sta per “Coprocessor”, infatti nei processori ARM oltre al processore principale ci dovrebbero essere dei coprocessori, da poter aggiungere o togliere in caso di necessità (ad esempio quello dei floating point serve solo se la macchina deve svolgere operazioni di quel tipo). Questi coprocessori sono quindi specializzati per cose che servono per alcune applicazioni ma non per tutte.

In Amber non è chiaro se esiste il Coprocessore per i floating point, e se esiste è l’unico coprocessore che ha. Altri coprocessori potrebbero servire per accedere a un’interfaccia grafica, o per trasferimenti di mma ecc.   
Il modo con cui il processore affida ruoli ai coprocessori è trasferendo una certa quantità di dati dai registri del processore a quelli del coprocessore e dopodiché inviargli delle istruzioni per lavorare su quei registri.

Le istruzioni CODTRANS sono dei data transfer che anziché operare sui registri della cpu lo fanno su quelli di un coprocessore (permettono il load/store tra RAM e un registro di un coprocessore). Il formato delle istruzioni è simile a quelle di tipo TRANS (si deve specificare sia l’indirizzo in RAM che il registro su cui operare).

Dalla codifica di queste istruzioni si nota che ci possono essere fino a 16 coprocessori perché in un punto c’è un campo da 4 bit chiamato “numero del coprocessore”.

Le istruzioni COREGOP sono le istruzioni che il processore principale ordina al coprocessore di eseguire. Un esempio tipico di operazione da outsourceare è il prodotto di due numeri in floating point.  
I vantaggi di questa organizzazione, oltre alla modularità, è il fatto di svincolare la pipeline della CPU principale dai tempi di esecuzione delle istruzioni del coprocessore (così mentre il coprocessore sta elaborando l’unità di esecuzione della CPU è libera e può eseguire istruzioni successive). Ovviamente, se l’istruzione successiva è nuovamente un’istruzione da affidare allo stesso coprocessore a questo punto la pipeline deve andare in stallo finché il coprocessore non finisce di eseguire almeno la prima istruzione (quindi il sistema funziona bene quando istruzioni di vario tipo sono mischiate tra loro).

Le CORTANS (R sta per Register) ha come particolarità di prendere l’input o l’output da registri della CPU principale anziché dalla memoria RAM (quindi permette di copiare registri tra processore e coprocessori).

L’ultima istruzione da trattare è l’istruzione Software Interrupt (SWI). Tutte le istruzioni, quindi anche questa, hanno i primi 4 bit di predicato di esecuzione (che corrispondono ai valori NZCV del “registro” di stato del PC).  
L’idea dell’istruzione è questa: si parte da un programma eseguito in modalità utente, chiamando SWI il processore si sposta in modalità supervisore (gli ultimi due bit del Program Counter passano da 00 a 11). Contemporaneamente viene fatta una modifica nel PC, inserendovi una certa cosa che non è stabilita dal programma che ha chiamata la SWI: viene inserito al suo interno un valore costante indipendente dai 24 bit precedenti. Tale valore è la prima istruzione del gestore delle interruzioni software (che sono le Trap), ossia l’indirizzo del Trap Handler. Così come nella Branch and Link il vecchio valore del PC viene prima copiato all’interno del registro LP (il registro 14). Quindi queste 3 istruzioni: cambio modo di esecuzione, branch and link e inserimento di un valore nel PC vengono effettuate in un’unica volta dall’istruzione SWI. Il programma precedente può essere ripristinato, ma solo dopo che il trap handler avrà terminato l’esecuzione.

Un processo analogo avviene con le richieste di interrupt, con la differenza che esistono sia l’interrupt normale che l’interrupt veloce. Anche queste due istruzioni, se non sono mascherate dai bit del registro di stato provocano le 3 istruzioni in una volta (passaggio a modalità ipervisore, salvataggio PC, modifica PC9.

Nel caso della SWI, il valore costante inserito è il valore 8. Quindi all’interno della memoria RAM, all’interno della cella di memoria 8 ci deve essere la prima istruzione del SWI Handler (o trap handler). Questa istruzione ovviamente è caricata in memoria durante la fase di bootsrap (quindi è il sistema operativo che determina l’handler per l’istruzione SWI).

Vediamo anche il vettore delle interruzioni. Sono previste un certo numero di interruzioni. La prima è chiamata RESET, che porta nel modo supervisore e usa l’indirizzo di RAM 0. In genere questo comando è dato quando si preme sul pulsante di accensione (più un altro pulsantino che però è nascosto) ed effettua un reset. L’handler che si trova all’indirizzo 0 sarà l’istruzione di bootsrap (o il BIOS in un portatile). Le altre interruzioni sono UND INSTR (istruzione non riconosciuta) ed è sostanzialmente una trap: quando il decoder riconosce un’istruzione come codificata male o non giusta da eseguire viene gestita in questo modo, l’indirizzo di questo handler è il 4. L’istruzione SWI porta anch’essa in modo supervisore e carica nel PC la costante 8. Poi ci sono altre due trap, la PREF(ETCH) ABORT e la DATA ABORT: portano anch’esse al modo di esecuzione 3 e corrispondono agli indirizzi 12 e 16. Dopodiché abbiamo un’altra trap che si chiama ADDRESS EXC (exception) che porta nel modo supervisore e carica nel PC la costante 20 e infine i due tipi di interrupt: l’Interrupt normale IRQ che porta allo stato 2 con indirizzo nel PC 24 e l’Interrupt Veloce, FIRQ, che porta al modo di esecuzione 1 con l’handler all’indirizzo 28.

Si nota quindi che le prime celle di memoria RAM non sono gestibili in maniera arbitraria, poiché devono contenere l’indirizzo della prima istruzione di un handler. La RAM per uso generale parte quindi dall’indirizzo 32 (e prosegue da lì in poi).   
Quello che abbiamo presentato è appunto detto Vettore degli Interrupt (anche se gli interrupt veri e propri sono solo il RESET e gli ultimi due elementi del vettore, IRQ e FIRQ, mentre gli altri sono Trap).   
Gli interrupt sono generati da qualcosa di esterno (un dispositivo o un pulsante o uno switch premuto), mentre le trap sono segnalazioni di situazioni anomale o errori presenti nel codice del programma in corso.

L’istruzione SWI (…1111…) è particolare perché non è segnalata da un dispositivo esterno ma è sollevata dal processore durante l’esecuzione di un programma (è una trap attivabile volontariamente). Tutte le altre segnalazioni vengono attivate/sollevate da componenti hardware che riconoscono la situazione anomala (seppur si tratti di un’anomalia del software).

La PREF ABORT, la DATA ABORT e la ADDRESS EXC sono sollevate dall’MMU. L’MMU non fa parte della definizione del processore (perché il processore non lo vede, mentre se è perfettamente in grado di vedere la Cache). L’MMU ci deve essere se vogliamo utilizzare il processore in versione virtualizzata (altrimenti possiamo solo eseguire codice in modalità 3). La scelta quindi di non includere l’MMU all’interno del processore è stata un po’ anomala (seppur abbia portato a una miglior progettazione dell’hardware). È infatti insieme al meccanismo delle Trap e delle istruzioni privilegiate che la traduzione degli indirizzi logici in fisici permette di virtualizzare il processore.

Avevamo parlato della SWI, che può essere eseguita dall’execute della pipeline. Se è vero il predicato di 4 bit si svolgono le tre operazioni: cambio modo, salvataggio PC, cambio PC; tutto in un ciclo di clock. Questo va specificato perché di registro 15 ce n’è uno solo ma di registri 14 ce ne sono 4. Se si cambia prima il modo di funzionamento allora il PC è salvato nel registro 14 (LP) del modo di funzionamento SVC (quindi il vecchio PC si troverà lì).

Tale LP conterrà il contenuto del registro PC incrementato di 8. Nel manuale dice che i 24 bit di offset del SWI non sono usati dal processore: sembrerebbero completamente inutili, ma invece servono per dare la possibilità al programma utente di mandare una segnalazione al Kernel del sistema operativo, mandato in esecuzione attraverso l’Handler dell’istruzione SWI. Cosa ci potrà mai essere dunque all’interno della cella 8? L’unica cosa che ha senso è un’istruzione di tipo BRANCH (perché le celle successive sono dedicate agli altri handler).  
Ipotizziamo che si vadano ad utilizzare gli indirizzi a partire dal 100, per l’handler della SWI. La BRANCH in 8 dovrà quindi essere codificata con i 4 bit di predicato a 0 (0000 = esegui sempre), i bit operativi (101), il link a 0 (0) e l’offset di valore positivo (perché la cella della RAM da raggiungere ha indirizzo maggiore). Quando si raggiunge tale BRANCH il valore del PC sarà aumentato di 8 (perché sarà stato eseguito il fetch della cella 12 e della 16), quindi il valore da sommare alla cella 8 (nel nostro esempio 92) va diminuito ulteriormente di 8 (quindi l’incremento finale in questo caso sarà 84).

Tipicamente il sistema operativo vorrà capire per quale motivo un programma avrà voluto mandare in esecuzione la SWI. Il motivo viene spiegato dai 24 bit di offset della SWI. Però questi 24 bit non sono contenuti all’interno di un registro del processore, quindi il nostro handler per vedere quale valore c’era in quei 24 bit deve andare a cercare nella RAM (nella porzione relativa al codice). Allora l’handler, quando arriva ad eseguire la prima vera istruzione (la 100, nel nostro esempio) può beccare quei 24 bit andando a vedere il contenuto del registro 14, perché contiene il vecchio PC, che punta all’indirizzo della cella di memoria che conteneva la SWI incrementato di 8. Quindi il nostro handler dovrebbe contenere una sequenza di istruzioni che: prenda il contenuto del registro 14 e gli tolga la costante 8, in modo da ricostituire il valore del PC al momento del fetch dell’istruzione SWI, poi un’istruzione di load dalla memoria RAM a un registro, poi un’operazione di azzeramento del primo byte (che conteneva il predicato e il codice operativo della SWI). Così si ha ottenuto all’interno di un registro il contenuto dei 24 bit dell’offset della SWI. In questo modo è possibile distinguere tra 2^24 motivi diversi per aver chiamato la SWI: così è come vengono implementate le system call.

Una system call potrebbe essere “perfavore allargami il segmento heap”, in modo da continuare la chiamata di un’istruzione di tipo malloc/realloc, verso il sistema operativo, che gestirà attraverso la non definita memoria virtuale la modifica delle tabelle di paginazione e di segmentazione per allargare il segmento heap.

Ci sono tantissimi (2^24) tipi di system call: estensione dell’heap, estensione dello stack, apertura di un file, ecc. Questo perché secondo il primo principio di Denning per poter fare un’operazione che richiede i privilegi del nucleo del sistema serve chiedere al sistema operativo. A parte questi 24 bit, il resto è lasciato al nucleo del sistema operativo (e all’handler della SWI). Questa istruzione, inoltre, ci permette di effettuare il cambio di contesto in accordo al principio del minimo privilegio.

Per poter riprendere l’esecuzione del programma dopo l’esecuzione della SWI è necessario innanzitutto che il sistema operativo dia una risposta positiva alla richiesta della SWI e dopo di ciò chiamare un’istruzione di ritorno, che rimetta il contenuto di LP nel PC e faccia ritornare la modalità di funzionamento del processore da modalità privilegiata a modalità utente.

Vediamo una delle caratteristiche che differenziano il processore ARM. Infatti, esso ha i 4 bit di predicato a ogni istruzione, mentre quasi tutti gli altri hanno la possibilità di rendere condizionali soltanto i salti. Il motivo per cui i processori ARM, e quindi il nostro processore Amber ha questa caratteristica è che rende più compatto, leggibile ed efficiente il codice.

Facciamo finta di scrivere un programmino in C per il calcolo di un massimo comun divisore tra due unsigned int, utilizzando l’algoritmo di Euclide. Abbiamo:

unsigned gcd(unsigned a, unsigned b) {

while(a != b) {

if(a > b)

a = a – b;

else

b = b – a;

}

return a;

}

Se implementiamo questo algoritmo su un processore ARM possiamo tradurlo così (immaginando che ci sia un assembler)

gcd: CMP R0, R1

SUBGT R0, R0, R1

SUBLT R1, R1, R0

BNE gcd

MOV R2, R0

MOV PC, LP

Notare che in questo caso il numero di istruzioni è molto vicino a quello della versione ad alto livello (perché le istruzioni di sottrazioni e di “ciclo” sono condizionali): questo permette di avere un codice estremamente compatto per essere di basso livello.  
SUBGT è l’operazione condizionata dal fatto che sia stato calcolato precedentemente un valore maggiore di 0: quindi se il CMP dice che R0 = R1 SUBGT non viene eseguita, altrimenti sottrae R1 a R0 e mette il risultato in R0 (tutto questo in una riga di codice).   
SUBLT, come suggerisce il nome, fa l’esatto contrario; viene eseguita una sottrazione (che sottrae R0 a R1 e mette il risultato in R1) solo se è stato prodotto precedentemente un valore minore di 0 (e anche in questo caso può venire prodotto dal CMP: si ha una situazione di tipo if/else if).  
BNE vuol dire invece BRANCH se NON UGUALE A ZERO, quindi se il CMP non ha prodotto 0 (e quindi R0 != R1) l’esecuzione salta nuovamente a gcd (per rieffettuare il controllo e le sottrazioni).  
Le ultime due istruzioni sono delle Move; La prima prende il contenuto di R0 e lo mette in R2 e la seconda ripristina il contenuto del PC con quello che trova all’interno di LP (queste due operazioni costituiscono il return).

Si è scelto di usare un Registro diverso (R2) per ritornare il valore, in realtà si sarebbe anche potuto utilizzare il registro R0 o R1, ma si è preferito fare così.

CMP è La classica istruzione di COMPARE che butta via il risultato della sottrazione e va a modificare i bit del registro di stato (in particolare ci interessano NZ: se sono 00 vuol dire numero positivo, se sono 10 vuol dire numero negativo, se sono 01 vuol dire che è stato generato 0 e quindi R0 = R1).

L’unica differenza sostanziale dal codice assembler che abbiamo presentato e il codice macchina è che non abbiamo indicato un indirizzo costante per l’operazione di salto ma abbiamo lasciato che se ne occupi l’assemblatore dicendogli di calcolarselo in modo che BNE produca un salto all’istruzione CMP R0, R1. Se volessimo abbassare ulteriormente il livello dovremmo inserire -20 al posto di gcd dopo BNE.

Se non si avesse la possibilità di rendere condizionali le SUB sarebbe necessario implementare dei salti condizionali (per implementare if/else), ma i salti condizionali causano lo svuotamento della pipeline: il programma finale non sarebbe quindi solo più complesso, ma anche meno efficiente.