Oggi riprendiamo la presentazione del processore Amber 23. Avevamo parlato della Pipeline su 3 livelli e dell’organizzazione della memoria. Poiché il program counter ha utilizzabili 2^24 bit i programmi hanno disponibili al massimo 2^24 parole di istruzioni da 4 byte. Se si inserisce una RAM a più di 2^26 Byte di memoria le celle in più potranno contenere soltanto dati e non istruzioni. Questa è un’anomalia di questo processore rispetto agli altri utilizzati al giorno d’oggi: è un esempio atipico di processore.   
Un’altra peculiarità di questo processore è quella di avere una istruzione condizionale. Quindi tutte le istruzioni vengono codificate con un predicato di 4 bit iniziali, che costituiscono la condizione che se è vera fa sì che l’istruzione possa essere eseguita (se è falsa in quel ciclo di clock non eseguirà niente e il ciclo andrà perso). Questa cosa ha una dubbia utilità: perché mai uno dovrebbe scrivere le istruzioni in questo modo? In realtà questa è una ottimizzazione, il motivo lo vedremo dopo.

Tutte le istruzioni usano lo stesso modo di indirizzamento. È una architettura di tipo RISK, infatti ha un insieme di istruzioni di tipo Load/Store: usa la memoria per caricare un valore in un registro o per salvare il contenuto di un registro, mentre le operazioni di tipo aritmetico sono basate sul contenuto dei registri. Non esistono solo le funzioni di Load/Store, ma anche la Load/Store multipla, che permette di salvare o caricare dalla memoria un numero arbitrario di registri.

Analizziamo la codifica delle istruzioni. Si usa suddividere le istruzioni in tipi, che sono stati definiti come:

* Regop: comprende tutte le istruzioni che fanno uso dei registri interni (si tratta delle operazioni di tipo aritmetico-logico). Queste istruzioni sono composte con i primi 4 bit che rappresentano la condizione di esecuzione, nei 28 rimanenti i primi 2 bit devono assumere il valore 0 (00 è l’identificatore delle Regop), i successivi bit ci dicono che istruzione è e con quali operandi. Il bit 25 viene chiamato “I” e serve per specificare/distinguere l’indirizzamento tra immediato e non. I bit dal 24 al 21 sono il codice operativo dell’istruzione (per distinguere somma da sottrazione da funzioni logiche ecc.). Il bit in posizione 20 viene chiamato bit “S” e permette di distinguere se si vuole modificare il flag della condizione (S=1 significa cambia i Flag, che vengono utilizzati per verificare la condizione di esecuzione delle istruzioni successive, S=0 significa non cambiare i flag indipendentemente dal risultato dell’istruzione corrente. I Flag si ricorda che sono i bit di stato del PC chiamati NZCV). Dopo il bit S abbiamo 4 bit tra il 19 e il 16 che indicano uno degli operandi (Rn), nelle posizioni da 15 a 12 si ha un altro operando (Rd). Il registro d (Rd) fa da destinazione, mentre Rn fa da addendo/fattore. I bit rimanenti, da 0 a 11, sono quello che viene chiamato “Shifter operand” (nel processore c’è un meccanismo molto efficiente per le operazioni di scorrimento, che possono essere fatte in un solo ciclo di clock), quindi il secondo operando dipende dal bit I e dai 12 bit di shifter operand che specificano il valore che dovrebbe essere preso come operando (nel caso di una somma farebbe da secondo addendo). Facciamo per esempio finta che l’indirizzamento sia immediato e si voglia inserire in R1 il contenuto di R0 più 5: nei 4 bit di condizione si metterà il codice per dire di eseguire sempre, poi bisogna mettere 00 (regop) poi bisogna indicare che l’indirizzamento è immediato, poi bisogna inserire il codice dell’operazione di somma, poi nel bit di flag ci va 0 e nei bit dei registri ci vanno 1 e 0 (perché R1 fa da dest e R0 fa da addendo) e nello shifter operand ci va codificato il numero 5 (0…0101).  
  Nel documento PDF (Manuale) La tabella 8 fa distinzione tra 3 modi di indirizzamento diversi: il primo è quello immediato (32 bit). L’idea è quella di costruirsi la rappresentazione di un numero su 32 bit partendo dai 12 bit del shifter operand (quindi bisogna espandere). Il modo più semplice sarebbe prendere lo shifter\_op, inserirlo nei bit meno significativi e completare la rappresentazione con tutti 0 (così si possono rappresentare numeri piccoli ma facilmente, ciò andrebbe bene per codificare le costanti come 5), ma il progettista del processore si è voluto complicare la vita e ha provato a rappresentare numeri più grandi usando solo 12 bit: i trucchi che usa sono spiegati nelle tabelle e 9. Per quanto riguarda i valori immediati, essi prevedono che il bit I (25) ha valore 1. Per questo motivo, in questo caso, i 12 bit di shifter op vengono suddivisi in 2 parti, una prima di 4 bit e una seconda parte di 8. Questi 8 bit vengono chiamati Imm\_8, mentre i 4 bit più significativi vengono chiamati Encode\_Imm. Con l’encode\_imm viene scritto un numero compreso tra 0 e 15, che viene usato per indirizzare un elemento della tabella 9, che ci dice come usare i successivi 8 bit per costruire la rappresentazione su 32 bit. Quando lì si trova il valore 0 succede il caso più semplice, si prendono gli 8 bit meno significativi dello shifter op, e si agganciano a sinistra di essi 24 0. La costante 5 è quindi composta da 4 zeri nei bit più significativi e poi i valori 00000101. Questo non è l’unico modo di codificare una ostante. Se si inseriscono nei 4 bit il valore 1 allora bisognerebbe prendere i due bit più a destra degli altri 8 e si vanno a piazzare nelle due posizioni più significative della nostra costante a 32 bit, seguiti da 24 0 e dai 6 bit rimanenti degli 8 iniziali non tagliati, quindi 0001 00000101 non è il numero 5, ma è 2^30 + 0 + 000001.  
  Il nome shifter\_op deriva infatti dall’idea che si hanno degli shift dei bit.  
  Se nei primi 4 bit si trova il valore 2 si prendono gli ultimi 4 bit e si fa la stessa cosa di prima con quei 4 (si buttano dalle cifre più significative del numero a 32 bit, ci si piazzano i soliti 24 zeri e alla fine si riportano i 4 bit non utilizzati degli 8 di partenza).  
  Andando avanti abbiamo rotazioni più grandi, quella su 8 su 10 e 12 bit (ecc.) quindi ci troveremo i nostri bit spezzati e in posizioni diverse. Quella finale è la rotazione di 30 posizioni (che è equivalente a uno shift a sinistra di 2).  
  La speranza di chi ha progettato tutto questo ambaradam è che il livello di precisione necessario non sia mai troppo alto.  
  Cosa succede però con I = 0? Significa che non è un indirizzamento immediato ma che fa riferimento ad un registro. In quel caso lo shifter\_operand è quindi un indirizzo. Abbiamo i nostri 12 bit, gli ultimi 4 sono un numero che rappresenta un registro (Rm), mentre gli altri 8 codificano per qualcosa di particolare. Poniamo per esempio di voler fare R1 <- R0 + R2, dopo aver codificato r2 nei 4 bit finali dello shift-op gli altri 8 bit vanno posti a zero (vogliono dire nessuna modifica). Se invece gli altri 8 bit non sono a 0 vuol dire che sui bit del registro indirizzato ci si vuole fare un’operazione di scorrimento. Quindi si può codificare anche (R1 <- R0 + (R2 << 3)) oppure (R1 <- R0 + (R2 >> 2)). Questo è dato dalle configurazioni diverse da zero nello shifter operand. Il bit 4 indica se lo shift deve essere immediato o avvenire prendendo il valore di un registro, i bit 5 e 6 servono per codificare il tipo di scorrimento (ce ne sono 4 tipi diversi: LSL = 00, LSR = 01, ASR = 10, ROR = 11). L sta per Logico, A sta per aritmetico (LSL è uno scorrimento logico verso sinistra e LSR verso destra), quindi i bit non vanno trattati come rappresentazione numerica e il bit più a sinistra non è considerato bit di segno, i bit che mancano vengono rimpiazzati dallo 0. Lo scorrimento di tipo aritmetico (A) è diverso, perché si deve tenere conto del segno nello scorrimento verso destra (e aggiungere a sinistra un 1 se il bit di segno era 1 e 0 altrimenti). ROR indica la rotazione: si ricicla quello che avanza a destra e lo si butta a sinistra.  
  I bit da 7 a 11 dicono, infine, di quante posizioni fare lo scorrimento (si può codificare da 0 a 31, 0 indica il tieni il valore originario). Si può tuttavia far sì che tale scorrimento non sia appunto, diretto, ma che avvenga anch’esso tramite registro (come indicato dal bit 4), quindi si può codificare l’operazione R1 <- R0 + (R2 << R6), questo avviene quando il bit 4 presenta il valore 1. Il nostro registro per fare lo scorrimento lo chiameremo Rs ed è indirizzato dai bit che vanno da 8 a 11. Il motivo per cui c’è solo la rotazione verso destra e non a sinistra è perché tanto possiamo ruotare fino a 31 posizioni.  
    
  Vediamo adesso il codice operativo di alcune istruzioni. Il valore 0 corrisponde alla AND, il valore 1 all’OR esclusivo, il valore 2 alla SOTTRAZIONE e il valore 3 alla SOTTRAZIONE inversa (perché non è commutativa, quindi il codice 2 indica Rn – shifter\_operand mentre 3 indica shifter\_operand - Rn). Il valore 4 indica la SOMMA e il valore 5 la SOMMA COL RIPORTO (che modifica il flag C del registro di stato).   
  L’istruzione di TEST ha come scopo quello di modificare i flag ma non produce alcun risultato. Cioè non viene preso in considerazione il campo Rd. Si prende Rn e si mette in AND con lo shifter operand, producendo come risultato il flag NZC, però non si memorizza da nessuna parte il risultato ottenuto (serve solo per il cambiamento dei flag). C’è un altro Test col codice 9 che fa la stessa cosa con lo XOR anziché con l’AND.   
  L’istruzione 10 è la COMPARE e fa una cosa analoga ma con l’operazione di sottrazione (Rn – Shifter\_Operand), che è analoga alla 11 COMPARE\_NEGATED che però fa la somma. Lo scopo di queste istruzioni è solo di modificare i flag.  
  Il numero 13 codifica l’istruzione MOVE, che prende il contenuto di Shifter Operand e lo salva in Rd (Rn non è preso in considerazione).  
  Abbiamo poi la BIT CLEAR e la MOVE NOT, varianti della funzione AND e della funzione MOVE.  
  La MOVE oltre a manipolare i dati potrebbe essere usata per modificare il flusso di esecuzione delle istruzioni (inserendo come Rd 15, ossia il Program Counter), quindi può fare anche da JUMP.