Oggi vedremo un esempio di processore e il suo utilizzo. Si chiama Amber. Esso fa parte della famiglia di processori ARM (che sono quelli presenti dei dispositivi mobili), un’architettura nata negli anni 80 e della quale sono uscite numerose versioni. La loro proprietà intellettuale è di una ditta, quindi tutti i costruttori di dispositivi mobili devono pagarne la licenza. La particolarità della versione Amber è che non è coperta de nessun brevetto, quindi è disponibile a tutti, e parte da ARM2 (in realtà la proprietà intellettuale c’è ma è open-source). La licenza ad esso associata è la licenza LGPL. Quindi possiamo usare il manuale senza dover pagare i diritti di autore o sviluppare versioni diverse.  
L’implementazione di questo processore è basata sulla tecnologia delle FPGA. Le FPGA vengono in genere realizzate per l’implementazione fisica dei prototipi (Una FPGA può costare qualche centinaio di euro).

Essendo derivato da una versione di ARM degli anni 80 è un po’ obsoleto, tuttavia l’architettura è molto simile a quella dei processori moderni, perché l’unica differenza è la velocità di clock e il numero di transistor.

Segue un’architettura RISC, quindi segue la prassi di occupare pochissimo spazio, tuttavia la semplificazione (meno transistor) si paga in termini di prestazioni.  
Ci sono due versioni di questo processore, la Amber 23 e la Amber 25. Sono quasi identiche a livello di programmazione, ma le dimensioni e le prestazioni della Amber 25 sono leggermente maggiori e migliori. La cifra delle decine indica la Versione di ARM da cui si è partiti (in questo caso ARM2) mentre il numero delle unità indica gli stadi della pipeline (l’unità 23 ha 3 stadi, la 25 ne ha 5). Entrambi questi processori perfettamente funzionanti.

Noi inizieremo vedendo la Amber 23.

Il prof sta scrivendo un programma di simulazione di questo processore.

Iniziamo dalla architettura. Abbiamo una pipeline a 3 stadi. Queste 3 componenti effettuano fase di fetch, decode e execute delle istruzioni. Integrato nel processore troviamo anche una cache di primo livello. Il dispositivo che fa il fetch non è connesso col bus dati ma è connesso con la cache. Questa cache è modulare, per poter avere implementazioni diverse caratterizzate da un costo elevato in confronto alle prestazioni. Il livello di associatività può variare tra 1 (economica) e 8 (costosa). La caratteristica di questa cache è che non differenzia istruzioni e dati. È connessa non solo all’unità di fetch, ma anche all’unità di esecuzione. La fetch va soltanto in lettura nella cache, mentre l’execute può effettuarvi sia lettura che scrittura.  
Il sistema è progettato in modo da funzionare regolato dalla tempistica data da un clock, dimensionato in modo da permettere un’esecuzione in base alla velocità di accesso alla memoria cache (se i dati sono nella cache le istruzioni sono eseguite il più velocemente possibile). Se i dati non sono nella cache si ha un rallentamento nell’esecuzione, che viene ottenuto mandando in Stallo la Pipeline (ossia bloccando il trasferimento dei dati da uno stadio all’altro). Ciascun modulo si occupa di un’istruzione a un x istante di tempo; man mano che la macchina va in esecuzione le istruzioni vengono gestite in parallelo (a partire dalla 3 istruzione) e se tutto va bene è eseguita un’istruzione per ciclo di clock. La Pipeline può però, appunto, andare in stallo e provocare il blocco della coda. (anticipazione: nella Amber 25 i primi 3 stadi sono identici e ci sono 2 stadi aggiuntivi, uno per completare la scrittura in memoria e un quinto per completare la lettura dalla memoria, servono a evitare lo stallo degli altri tre stadi. Un’altra differenza della variante 25 è che la cache è divisa in due parti, ce n’è una che si occupa solo del codice ed è connessa solo alla fetch e un’altra che si occupa solo dei dati ed è connessa solo alla execute).   
Nella cache c’è un protocollo di consistenza che si attiva in caso di miss in lettura e che fa sì che l’indirizzo richiesto dal processore sia caricato in cache (anche sostituendone un altro). Per le operazioni di scrittura, queste sono gestite col Write Through (cioè scrivi il valore direttamente nella memoria RAM -> provoca un leggero rallentamento, perché costringe a mantenere la consistenza). D’altra parte è il protocollo più semplice da implementare.

Nel manuale c’è tutto tranne l’implementazione del decode, che è descritto come l’implementazione di una macchina a stadi (?). L’unità di decodifica non è stata descritta perché la codifica delle istruzioni non è affatto semplice.

Venendo alla microarchitettura, ci sono 16 registri visibili dal processore, alcuni di uso generale ( i primi 8), mentre altri hanno funzione specifica. Sono registri in 32 bit (anche se le versioni più recenti sono passate a 64 bit) che possono essere usati liberamente dal programma in esecuzione. La maggior parte delle istruzioni fanno riferimento al contenuto di questi registri: ad esempio l’istruzione di somma, che permette di definire tre registri, quello di destinazione (rd) e i due sorgenti (rn e shifter\_operand), come istruzione tipo potremmo avere ADD R3, R4,#1 (prendi #1 e R4 e inserisci la loro somma in R3).

Dei registri specializzati hanno degli alias diversi. R8, R9 e R10 vengono chiamati così, mentre R11 viene chiamato FP, R12 IP, R13 SP, R14 LP, R15 PC. Cominciamo dal più complicato, il PC (Program Counter), che è complesso per come viene implementato (ci sono delle ottimizzazioni che sembrano fatte da un genovese); infatti contiene sia il Program Counter che il Registro di Stato (servono perché alcune istruzioni possono avere un predicato che le blocchi se i valori precedenti rispettano certe condizioni). I 32 bit del registro sono suddivisi in 4 parti: i 4 bit iniziali danno lo stato dal punto di vista delle istruzioni precedenti (i risultati delle 4 uscite ausiliarie della ALU). Questi 4 bit sono chiamati rispettivamente N,Z,C,V (N=numero negativo == 31 bit del valore prodotto dal calcolo precedente, Z = zero == è stato prodotto 00..00, C=riporto/carry, V= overflow). Dopo questi 4, sono presenti altri 2 bit: I,F, essi sono i bit di maschera delle interruzioni (le I sono le Interrupt, le F le fast interrupt, quindi a priorità più alta), con 00 si disabilitano tutte le interruzioni e con 11 le si abilitano entrambe. Abbiamo poi 2 bit particolari alla fine, che sono il modo di esecuzione del processore (abbiamo parlato di istruzioni privilegiate che distinguono, anziché tra utente e sistema, in 4 modi: 11->Supervisor, 10->Interrupt, 01->Fast Interrupt, 00->User). Questo provoca l’effetto, oltre che stabilire il livello di privilegio (in modalità supervisor si possono fare cose non possibili nella modalità user), di gestire in maniera alternativa l’insieme dei registri: mentre i primi 8 sono registri normali, quelli successivi sono replicati. La versione del registro da prendere dipende dall’attuale modo di funzionamento del processore. Questo permette di avere gestori di interruzioni e gestori di trap più veloci: questo perché se si passa dalla modalità user alla fast interrupt cambiano anche i registri che vanno da 8 a 14, quindi il gestore delle interruzioni veloci non deve scrivere nello stack i contenuti dei registri (e ripristinarli prima della ripresa dell’esecuzione) perché va a lavorare su una copia del registro X che non è collegata al registro effettivo. Usando questo trucco si possono evitare diverse push e pop (che funzionano alla velocità della RAM anziché del processore).

I registri che vanno da 8 a 12 sono duplicati (o meglio, sostituiti) solo dallo stato di Fast Interrupt. In modalità user si vedono tutti e 16 i registri, però in modalità Fast Interrupt i registri che vanno da 8 a 12 sono una versione tutta loro, chiamata R8\_firq R9\_firq, …, R12\_firq. Il programma che gestisce le interruzioni veloci è tale quale a quello delle interruzioni normali, però lo swap dei registri permette di fare le cose più velocemente (NOTA: A livello hardware non ci sono 16 registri, ma di più, perché quelli da 8 a 12 sono duplicati).

I registri 13 e 14 sono privati per ogni modo di esecuzione, abbiamo anche R13\_rq, R13\_sdc, R13\_?, e le versioni equivalenti per R14 (ci sono 4 copie sia del registro 13 che del 14). Il numero totale di registri è quindi 29, ma in ogni momento il programma ne può accedere soltanto 16.

Riprendendo il PC, rimangono 24 bit che sono effettivamente il program counter, quindi può indirizzare 2^24 celle di memoria RAM.

La RAM è organizzata in parole da 32 bit. Tutte le istruzioni sono quindi codificate su 32 bit. Nella fase di fetch bisogna accedere a una cella di memoria e recuperare quei 32 bit per decodificarli: tutte le istruzioni hanno uguale lunghezza. Dal punto di vista della memorizzazione dei dati il processore ammette de tipi: il tipo word da 32 bit e il tipo byte da 8 bit. Quindi l’indirizzamento della RAM viene fatto a livello di byte (i due bit meno significativi servono per identificare uno dei 4 byte all’interno di ogni parola). Es: la parola all’indirizzo 1000 necessità anche l’indirizzo del singolo byte per essere acceduta. Ci sono due alternative: Big Endian e Little Endian. Questo è un processore di tipo Little Endian (l’indirizzo del byte è attribuito ai bit meno significativi). Quindi nell’esempio di prima i valori da 1000 a 1003 rappresentano gli indirizzi dei byte nella stessa parola. I normali processori ARM la scelta della rappresentazione è data al programmatore, ma in questo caso non è così.   
Un’altra differenza è che l’insieme delle istruzioni si è evoluto nel tempo (perché è passata da una rappresentazione a 32 bit, poi è stata ridotta a 16 bit per far stare due istruzioni in ogni parola -> il vantaggio è che permette il fetch di due istruzioni contemporaneamente, ma dà meno libertà nelle istruzioni -> Tale sistema è chiamato Thumb). Un processore ARM ha due metodi di funzionamento diversi, uno con istruzioni a 32 bit e una a 16 bit: il passaggio da un funzionamento all’altro è dato dall’istruzione di JUMP.   
Gli ultimi due bit di indirizzamento della memoria RAM non sono usati dalla RAM, ma sono usati dal processore per accedere ad uno dei byte della cella di memoria: ecco perché possiamo sostituire i due bit meno significativi del program counter (perché lui accede a parole di memoria, che essendo multipli di 4 fan sì che quei due bit siano sempre a 0 -> li si può riciclare per altri scopi).

Ci sono quindi 2^26 indirizzi possibili, ma nel PC ci sono solo 24 bit di indirizzamento perché esso prende parole intere.

Il Program Counter non è replicato.

Tra gli altri registri strani abbiamo il registro LP, il registro 14. Esso esiste al fine di realizzare in maniera efficiente le interruzioni dei programmi. Può essere usato sia per le interruzioni vere e proprie (grazie alla sua copia) salvando il contenuto del program counter al suo interno (anziché nello Stack, cosa che rallenterebbe l’interruzione).   
Le interruzioni, infatti, consistono nel salvare il contenuto del PC nel registro LP e poi modificare il contenuto di PC per indirizzarlo al gestore delle interruzioni. Quando si deve ripristinare l’esecuzione, basta spostare il contenuto di LP in PC (->sono tutte operazioni fattibili in un solo ciclo di clock, perché non vi è lettura o scrittura in memoria RAM). Ovviamente questo sistema funziona una volta sola: se LP non è vuoto (ad esempio c’è già un PC salvato in LP perché c’è già stata un’interruzione) e si vuole re-interrompere il programma il valore di LP andrebbe salvato all’interno dello Stack. Quindi questo metodo funziona molto bene per passare dallo stato user a quello interrupt, ma non funziona altrettanto bene se c’è un’interruzione durante un’interruzione, oppure se il gestore delle interruzioni effettua una chiamata ricorsiva.  
Quando il programmatore vuole chiamare una funzione, può usare l’istruzione di salto, il BRANCH (B), di cui ci sono due versioni (B e BL, la prima modifica semplicemente il program counter, il BL = Branch and Link, fa, oltre quello che fa B, la copia del precedente indirizzo del PC nell’LP, quindi dà la possibilità di riprendere l’esecuzione del programma precedente).

Poiché del registro 14 ne esistono 4 copie, ogni modalità di funzionamento ha la copia che vede solo lui del registro LP. Quindi, ad esempio, il gestore dei fast-interrupt può richiamare una funzione usando il BL senza prima dover salvare il contenuto del registro LP nello stack.

Il registro 13 si comporta da Stack Pointer. Quindi non ha senso usarlo per altri motivi.

Il registro 11 è il Frame Pointer, che viene usato assieme al SP per definire/delimitare la memoria RAM dedicata al corrente pezzo di codice. Uno tra questi due fa da limite inferiore, l’altro da limite superiore. La differenza è che lo Stack pointer può essere modificato da istruzioni di Push e Pop, mentre il FP viene modificato solo dalle chiamate di funzioni.  
Questo registro permette di localizzare le celle di memoria nello stack mediante indirizzamento indicizzato (perché l’FP non cambia frequentemente posizione).

Il registro IP non è molto importante.

I registri dall’8 al 12 hanno la particolarità di essere usati per migliorare la gestione delle interruzioni veloci grazie alla copia che hanno visibile per lui (che gli permette di operare senza fuckuppare il programma in esecuzione). La cosa non si applica per i registri dal 7 allo 0.