La terza parte del laboratorio consiste nel modificare il vector.h e il vector.c per far sì che accetti anche come tipo di dato i float. L’unica difficoltà è che le funzionalità aggiuntive vanno testate e quindi bisogna costruirsi un main e modificare il makefile.

Il makefile è definito in due parti. In quella iniziale sono stabilite le macro, nella seconda le regole di produzione dei file. Le macro sono definite assegnando dei nomi a delle stringhe di caratteri (per esempio FILEINPUT = vector). Questo permette di usare l’operatore $ nella seconda fase (quella delle regole) per utilizzare le macro come delle variabili. Scrivendo $(FILEINPUT): si assegna al comando make FILEINPUT quello che segue nella definizione della regola.  
C’è una prima riga in ogni regola che stabilisce le relazioni di precedenza, se si ha ad esempio $(F1): $(F2) $(F3) vuol dire che il file F1 dipende da F2 e F3: a questo punto bisogna verificare se F2 e F3 esistono. Se F1 non esiste ma F2 e F3 sì allora F1 viene prodotto utilizzando F2 e F3. Se esistono già tutti e tre bisogna controllare le date di ultima modifica: se F2 e F3 sono stati modificati più recentemente di F1 allora F1 viene rifatto. La seconda riga nella definizione di una regola DEVE iniziare col tab e ci sono scritti i comandi di shell necessari per generare il file F1. Per esempio:

vector.o: vector.h vector.c

gcc -c vector.c

Completiamo il discorso interrotto settimana scorsa.

Avevamo visto l’utilità del predicato di 4 bit delle istruzioni del processore Amber con un programmino di calcolo del massimo comun divisore (tramite algoritmo euclideo). Sfruttando i predicati il numero di istruzioni in codice assembly è molto vicino a quello di alto livello in c.

Gcd: CMP R0, R1

SUBGT R0, R0, R1

SUBLT R1, R1, R0

BNE gcd

MOV R2, R0

MOV PC, LP

SUB non modifica il registro di stato, mentre SUBS sì. Nel caso delle istruzioni di salto è complicato tenere conto dello stato del PC per via della Pipeline (infatti dovrebbe essere a +8 rispetto al momento in cui si è fatto il fetch dell’istruzione di Branch). Tuttavia, i Branch sono relativi, quindi per saltare indietro di 3 istruzioni bisogna inserire -20. Questo offset è però generalmente calcolato dall’assemblatore/compilatore. I processori diversi da ARM permettono di condizionare soltanto le istruzioni di salto: un processore di questo tipo richiederebbe di implementare in maniera più complicata la struttura di controllo di tipo if-else.

In tal caso otterremmo un codice simile a questo:

gcd: CMP R0, R1

BE end

BLT else

SUB R0, R0, R1

B gcd

else: SUB R1, R1, R0

B gcd

end: MOV R2, R0

MOV PC, LP

Le istruzioni sono chiamate con lo stesso nome delle operazioni in Amber ma utilizzate come funzionerebbero se non avessero il predicato. Il codice in questo caso presenta 3 righe di codice in più (che su 6 righe dell’altra versione costituiscono un aumento di lunghezza del 50%) ed effettua molte più istruzioni di salto (può arrivarne ad effettuare 2 per ciclo). Il problema di questo tipo di codice è che ogni salto fa svuotare la pipeline, quindi effettuare più salti fa perdere cicli di clock.

Gcd, else, end nei nostri esempi sono delle etichette ( o label) che aiutano noi in lettura e l’assemblatore durante la fase di compilazione. Volendo si potrebbero eliminare le label dalle istruzioni di salto e indicare precisamente di quanto modificare il PC. Tuttavia questo procedimento può produrre errore, quindi si usano le label e si scarica il calcolo sull’assemblatore. È analogo a quanto succede con i linguaggi di programmazione anche a basso livello, in cui istruzioni come MOV, B, SUB, CMP sostituiscono i codici operativi delle istruzioni.

Si può fare un commento sul Branch relativo. Storicamente sono sempre esistite due forme di salto per i processori: il salto relativo (in cui valore da sommare al PC viene indicato) e il salto assoluto (in cui è indicato l’indirizzo esatto). Nel caso di Amber c’è solo l’istruzione di salto Relativo, ma è comunque possibile raggiungere un qualsiasi indirizzo di memoria tra quelli destinati al codice (probabilmente grazie al fatto che il PC indirizza su solo 2^26 indirizzi).

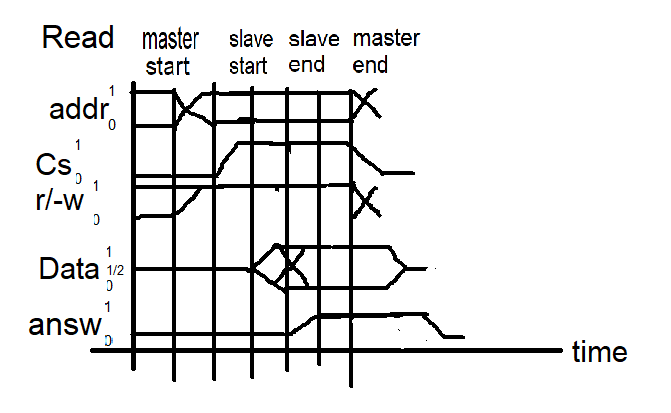
Il salto relativo ha il vantaggio di permettere la scrittura di codice rilocabile: non serve conoscere in quale posizione il segmento della RAM delle istruzioni si trova per poter accedere a una istruzione. Un secondo vantaggio è che permette di dover modificare soltanto il Program Counter senza dover studiare una codifica diversa se si volessero modificare le dimensioni della memoria. Con un salto di tipo assoluto se si modificasse la dimensione del PC (perché si espande la RAM) andrebbe anche modificato il valore dell’indirizzo passato all’istruzione di salto.

Piccola deviazione dell’ordine di presentazione degli argomenti: Parliamo dei protocolli per l’implementazione di un BUS.

Il BUS deve mettere in comunicazione RAM e CPU: tale comunicazione è però di tipo asimmetrico. Il processore è un’entità attiva, mentre la RAM è un’entità passiva. Il protocollo del BUS viene infatti detto Master/Slave.  
Gli ordini che può dare il processore sono tipicamente 2: lettura e scrittura.  
Abbiamo già brevemente parlato dell’interfaccia della RAM, che normalmente ha una serie di fili di indirizzamento, un paio di fili di controllo (Cs, r/~w) e i fili per i dati, che funzionano sia in ingresso sia i uscita (a seconda del tipo di operazione). Visto com’è fatto lo slave, anche il master, la CPU, dovrà avere in uscita dei fili per gli indirizzi, i due per i segnali di controllo e quelli per la parte dati, anch’essi gestiti coi dispositivi a tre stati.

Per poter tradurre queste interazioni in un protocollo vengono considerate due soluzioni diverse. Il protocollo può essere Sincrono o Asincrono. Un BUS Asincrono non ha bisogno di un segnale di clock, mentre un BUS sincrono sì.  
Vediamo prima il protocollo Asincrono. Consideriamo l’accesso in lettura e in scrittura e proviamo a definire un comportamento nel tempo da dover essere assunto dal dispositivo master e dallo slave per poter eseguire questa operazione.

Ciclo di lettura.

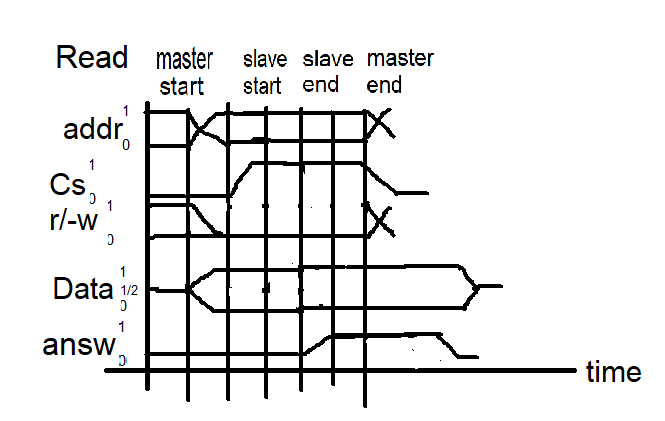


In un primo momento il ChipSelect deve rimanere a 0 perché non si è sicuri che gli altri fili abbiano già assunto i valori giusti. Solo dopo si può portare il valore di Cs da 0 a 1. La RAM slave, continua a guardare il bit Cs; non appena vede che è diventato 1 parte per eseguire l’ordine che le è stato impartito. Vede che R/-w è impostato a 1 e quindi capisce che è un ciclo di lettura: da quel punto in avanti lui connette i dispositivi a tre stati sulla parte dati. Poiché però anche la RAM ha bisogno di tempo per fornire alla CPU una risposta sensata è necessario introdurre un ulteriore filo di risposta che va dallo Slave al Master.

Quando il Processore vede di aver ricevuto risposta avvia la memorizzazione di quello che è presente sui fili dei dati. Quando tale memorizzazione è riuscita il processore toglie il valore 1 dal bit Cs e si può considerare conclusa l’operazione di lettura.

Poiché sia La RAM che la CPU hanno bisogno di “conferme” (o risposte) da parte dell’altro componente prima di procedere questo sistema funzionerebbe indipendentemente da quanto tempo impiegherebbero le operazioni per essere eseguite. Cs e Answ fanno infatti da messaggeri dello stato dei due componenti. Nei circuiti asincroni Answ è detto Ack.   
Una nuova operazione può essere avviata dal momento in cui il processore porta a 0 il bit di risposta.

Per il ciclo di scrittura:



Il grafico è quasi uguale, ma ad attaccare i fili tramite i dispositivi a tre stati è il processore e non la RAM (oltre che avere il bit r/-w a 0 ovviamente). Quindi vengono attaccati quasi subito, appena dopo che il processore si è assicurato di aver immesso sul BUS il valore giusto.

Il circuito asincrono funziona senza overlap o confusione tra i dati perché ciascun dispositivo tiene conto dei propri ritardi e aspetta perché finisca l’altro (quindi non importa la lunghezza del bus o la velocità della RAM).  
Lo svantaggio principale, però, è che c’è un grosso ritardo nella conclusione delle operazioni: si osserva infatti che dopo la slave end l’operazione è praticamente conclusa, tuttavia bisogna preparare il BUS per l’operazione successiva. Ciò provoca l’attesa di un tempo (per l’azzeramento dei bit Cs, anser e lo stacco dei fili per i dati dai dispositivi a tre stati) che è uguale al tempo che ci mettono le informazioni a fare due volte avanti e indietro lungo il BUS (che corrisponde al dover far azzerare i bit Cs e Ack). Questo presenta un limite, perché se il bus è lungo ciò limita incredibilmente la velocità del trasferimento dei dati. Al giorno d’oggi, anche se un BUS può arrivare a essere lungo pochi mm, è importante tenere comunque conto della dimensione, specie se si considerando le elevatissime frequenze di clock dei processori al giorno d’oggi, dove pochi nanosecondi fanno la differenza.

La soluzione a ciò consiste nel far fare ai bit un solo ciclo di andata e ritorno (indirizzo e risposta) ed evitare il secondo giro (reset di Cs e Ack). Questo secondo giro infatti è una necessità solo del protocollo asincrono (nel caso delle operazioni di scrittura potrebbe addirittura usare solo l’andata). Quindi si potrebbe passare, da un punto di vista teorico, da quattro a due e una unità di tempo. Ciò è possibile tramite i BUS con protocollo Sincrono.