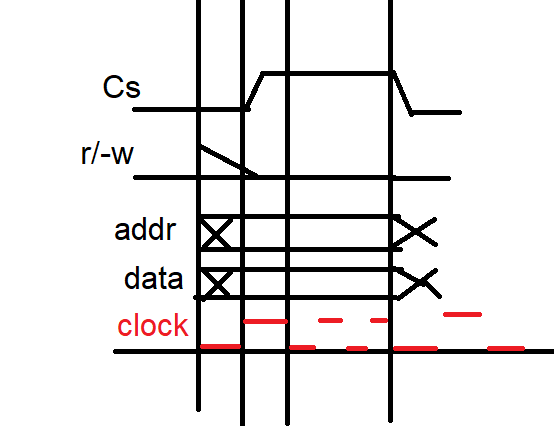
Abbiamo visto i protocolli per il Bus asincroni. Abbiamo visto che è facile da concepire e da realizzare, perché tutto avviene in modo automatico gestito da segnali di controllo: la sincronizzazione avviene in modo asincrono (?). I segnali mandati dal master sono il Chip Select e il segnale r/-w (quest’ultimo non c’entra però col protocollo asincrono, c’è in ogni caso). Lo Slave risponde al master con una Ck. Il protocollo asincrono prevede due round-trip completi (quando Cs passa da 0 a 1, poi Ck passa da 0 1, poi Cs passa da 1 a 0 e Ck ritorna da 1 a 0). Il numero di unità di tempo in questo caso è quindi 4.

Dal punto di vista concettuale ciò è però limitante perché per la lettura servirebbero solo 2 unità di tempo e per la scrittura 1 sola unità di tempo. Per questo si introduce il protocollo di tipo sincrono.

Il protocollo sincrono prevede che ci sia un segnale di clock (anche esterno) che viene mandato sia al master che allo Slave.

Il segnale di clock varia alternando 0 e 1 e quindi si comporta come un orologio a pendolo. Per poter introdurre un protocollo sincrono è necessario conoscere esattamente i tempi di propagazione dei segnali sul BUS e i tempi di esecuzione del master e dello Slave: tutto ciò è necessario perché è il clock a stabilire il passaggio dei dati sul BUS. Quando il clock passa da 0 a 1 il master può mandare i propri dati/il proprio segnale r/-w = 0 e imposta il segnale Cs a 1.

Nell’operazione di scrittura, poniamo il caso che dal Processore ci voglia mezzo ciclo di clock per inviare i dati e mezzo ciclo di clock per trasmetterli all’altro dispositivo. Poniamo inoltre che ci voglia un ciclo di clock e mezzo per portare a termine l’operazione di scrittura (dopo un primo mezzo ciclo perso perché: ragioni? Forse deve partire dal valore 1 o forse serve mezzo ciclo per rendersi conto che i dati sono arrivati, boh) e dopo di ciò il dispositivo Master può azzerare il valore Cs, togliere l’indirizzo e staccare il dispositivo a tre stati.



Come si nota ciò permette di risparmiare diverse unità di tempo poiché vengono impiegati pochi cicli di clock, inoltre non è necessario il segnale Ck. Tuttavia per poter definire questo protocollo è necessario sapere quanto è necessario in termini di tempo per poter terminare ciascuna fase delle operazioni. In un sistema vero anche nelle operazioni di scrittura c’è sempre un segnale di ritorno verso il master perché anche se si perde un po’ di efficienza fa sì che si possano collegare anche dispositivi Slave a velocità diversa. In questo caso però il segnale di ritorno si chiama WAIT anziché Ck (e serve solo se lo slave non è riuscito a terminare in tempo l’operazione, infatti come suggerisce il nome questo segnale è una richiesta di attendere da parte dello Slave). Questo è il motivo per cui si preferisce usare i protocolli sincroni rispetto a quelli asincroni: sono più veloci e anche se lo slave fosse troppo lento rispetto alle aspettative si può adattare.

Il ciclo di lettura è praticamente analogo, con l’unica differenza che c’è anche una fase di ritorno dei dati.

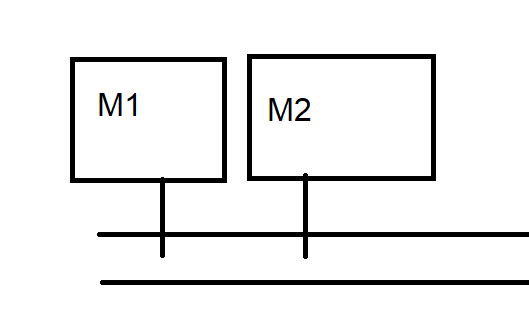
Terminiamo la trattazione dei BUS e introduciamo un’altra idea.

Un’ ulteriore complicazione che si può verificare nei sistemi reali quando progettiamo un BUS. I dispositivi che possono essere connessi al BUS sono più di due: tipicamente il BUS ha una struttura che permette di connettere in certi punti altri dispositivi attraverso i connettori.

Se continuiamo a inserire i dispositivi di tipo Slave non c’è nulla di particolare da notare se non che lo spazio di indirizzamento deve essere suddiviso tra questi Slave: se ci sono due moduli di RAM che contengono un Gb l’uno, un modulo deve rispondere a un sottoinsieme di indirizzi e l’altro a un sottoinsieme di indirizzi diversi. L’idea è di prendere i fili di indirizzamento e suddividerli a un certo punto. I bit più significativi vengono quindi utilizzati per raffinare il Chip Select (?). Infatti questo segnale passa attraverso un AND e si effettua un controllo tra la parte maggiore dell’indirizzo mandato dalla CPU e l’indirizzo dello Slave, se coincidono il segnale Cs viene mandato a quello Slave, altrimenti si passa al controllo dell’indirizzo del dispositivo successivo.

Se però io volessi aggiungere un altro dispositivo Master? In questo caso è più complicato da gestire, perché il dispositivo Master ha l’iniziativa e non pensa di dover competere con altri dispositivi Master.  
In questo caso il Bus deve in qualche modo prendere l’iniziativa e dare la possibilità di agire in ogni momento a soltanto uno dei due dispositivi master. La logica che permette ciò è detta Logica di Arbitrazione del BUS. Il Master quindi rimane padrone degli Slave, ma non è padrone del Bus, il Bus rimane indipendente e funge da arbitro in cui avvengano due richieste in contemporanea di utilizzo del BUS. Anche la fase di arbitrazione può essere asincrona o sincrona, però viene generalmente realizzata in modo sincrono.

Quindi un dispositivo master che vuole usare uno slave deve mandare una richiesta all’arbitro; si aggiunge quindi una fase in più di richiesta di permesso. Normalmente l’arbitro può ricevere più di una richiesta, ne sceglie quindi soltanto una da esaudire con un criterio che sia rapido (in modo da non rallentare troppo le operazioni) e abbastanza consistente: quindi si introducono delle priorità. La priorità attribuita a ciascun master è determinata dallo slot del Bus in cui il master è inserito. Se vengono ricevute più richieste, quindi, il BUS basa la priorità sulla base di dove sono stati installati i master. Per esempio potrebbe dare sempre più priorità ai dispositivi più a sinistra.



(quindi in questo caso M1 ha più priorità di M2).

Il master che riceve il Grant avvia la propria operazione di lettura/scrittura, usando il bus come se ci fosse solo lui. Al termine della transazione il master rilascia il bus e prima di poterlo riacquisire ha bisogno di ottenere nuovamente il grant (quindi nel frattempo potrebbe essere usato da M2). Ciò crea un ritardo aggiuntivo, che può essere eliminato con una tecnica simile al Pipelining.   
Il tempo di arbitrazione, infatti, va in parallelo con una transazione: mentre un Master va in transazione altri master possono mandare la loro richiesta di accesso al Bus e il Bus può preparare le risposte (si forma una “coda” davanti al bus). Appena un Master finisce di usare il Bus, iniziano gli altri. In questo modo un Bus può permettere un funzionamento da parte di Master multipli con quasi nessuna penalizzazione a livello di tempi impiegati dal sistema per effettuare le operazione. Sul Bus le transazioni sono interrompibili, ma possono essere mischiate tra di loro (nel senso che in un certo periodo di tempo può esserci il master M1 a effettuare la scrittura e dopo un altro periodo di tempo può esserci M2 ad effettuare la lettura).

Perché può essere utile inserire più master in un sistema? Chiaramente almeno un master è necessario, poiché è il processore. Gli altri master possono però essere dei dispositivi di input output, che per questioni di efficienza possono avere la capacità di effettuare operazioni di lettura e scrittura sulla memoria RAM (tale capacità è detta DMA: direct memory access). Questi dispositivi chiaramente non sono paragonabili a un processore: la loro attività, anzi, dipende dal processore in quanto sono programmati da esso. Per questo motivo il dispositivo DMA può agire sia come master che come Slave (è uno slave quando il processore lo programma accedendo ai suoi registri mappati in memoria RAM e una volta programmato il dispositivo inserisce ed estrae dalla RAM le informazioni a lui necessarie o da lui raccolte come dispositivo Master).

Per ora interrompiamo il concetto di accesso alla memoria da parte dei dispositivi I/O, ci serviva solo per dimostrare che possiamo avere più di un dispositivo Master ad accedere al sistema.   
Se il processore trova il BUS già occupato, esso si deve fermare, quindi in questo caso la Pipeline deve andare in stallo.

Il processore ha sempre almeno un livello di cache, quindi le operazioni di lettura e scrittura avvengono sulla cache di primo livello, quindi in realtà è la cache che fa la richiesta alla RAM e che eventualmente viene messa in attesa (anche se una messa in attesa della Cache può causare nella maggior parte dei casi un’attesa nel processore).

Parliamo adesso del programma di simulazione del processore Amber. C’è un’interfaccia grafica con una finestra principale. In alto a destra ci sono la possibilità di mettere il nome di un file e un pulsante di load. Sotto di essi ci sono delle zone che fanno vedere alcune zone di memoria RAM: la prima fa vedere la zona a cui punta il PC, quindi che riguarda il fetch, mentre la seconda riguarda la parte dei dati che stanno venendo utilizzati (ogni zona fa vedere 7 celle). A sinistra ci sono i registri del processore (che sono in parte fissi e in parte commutabili): le parti commutabili vengono visualizzate in tutte le versioni (quindi si vedono due registri 8, due 9, due 10, due 11 e due 12, per le versioni normali e fast interrupt e quattro registri 13 e quattro 14, in cui ogni modo di funzionamento ha il suo paio di registri). Infine c’è il Program Counter che ovviamente contiene anche i flag e il modo di esecuzione perché fa anche da registro di stato. In alto a destra c’è un pulsante chiamato “tick” e un menu che fa scegliere tra next e prev: l’idea è di far mandare avanti il clock di un ciclo alla volta cliccando sul click (selezionando prev anziché next si può tornare indietro nel tempo). Abbiamo al centro i tre stadi della pipeline (fetch, decode ed execute): la fetch mostra la posizione in RAM a cui puntava il PC e il codice operativo dell’istruzione (in esadecimale). La decode mostra la descrizione testuale in assembler dell’istruzione e la execute mostra quale istruzione sta venendo eseguita e si occupa di mandare in stallo gli stadi precedenti quando l’istruzione impiega più di un ciclo di clock per essere eseguita.

Nel caso di istruzioni di salto, e quindi cambio del valore del PC bisogna mandare in stallo e svuotare la pipeline, facendo ripartire il fetch dal nuovo valore del PC al ciclo di clock successivo. Oltre ai file .py (il sorgente è scritto in python) ci sono dei file .run (euclide.run). Questi sono esempi di programmi che possono essere caricati in RAM e poi eseguiti da parte del processore.   
Se il processore è in fase 3 (supervisor) i bit di maschera delle interruzioni (sempre nel PC lol) sono azzerati (le interruzioni non partono): quando mandiamo in esecuzione un’applicazione quindi il sistema operativo deve innanzitutto impostare i registri. Il Frame Pointer viene quindi inzializzato dal sistema operativo a un certo valore (128), e il sistema operativo deve trovare in quella cella di memoria RAM i suoi dati statici. Quindi il file deve essere consistente e trovare uno dei suoi valori iniziali nel registro 128. Questa è una convenzione scelta dal prof quando si è posto il problema di come fa il sistema operativo di dare accesso alla memoria a una operazione. Inoltre bisogna reimpostare la modalità del processore a 0 prima di far partire l’applicazione. L’unico modo per caricare nel registro 15 il modo di funzionamento è quello di eseguire un’istruzione di load multipli: si inserisce in RAM il valore che si vuole inserire nel PC e poi usare il load multiplo per caricarlo nel PC: se si effettuasse un’operazione di move nel PC verrebbero modificati solo i 24 bit di indirizzo e non verrebbero modificati i bit di stato.

Alla fine dell’applicazione viene chiamata l’istruzione di ritorno, realizzata col MOVE del valore nel Link Pointer all’interno del Program Counter (senza dover modificare il modo di esecuzione). Alla fine il risultato sarà in R2, quindi il processore salverà il risultato in RAM e avviene la terminazione del programma. In C la terminazione avviene tramite la system call “exit()” a cui si può passare come valore un intero. Quando ciò avviene il sistema operativo libera le risorse che erano in esecuzione e fa partire altri programmi.

In Amber le system Call sono realizzate mediante la SWI, che contiene un offset rappresentante un intero compreso tra 0 e 2^24-1. L’istruzione SWI provoca il passaggio alla modalità supervisor e inserisce il valore 8 all’interno del PC. La risposta alla terminazione di un’applicazione è che il programma venga resettato e riparta da capo.

I file d’esempio son oeuclide0.run e euclide1.run: il secondo ha un’istruzione in più rispetto al primo. Aggiungendo tale istruzione in realtà si riesce a completare l’esecuzione del programma in un numero di cicli di clock inferiore rispetto alla versione più lunga.

Questo vuol dire che compilatore può fornire più scelte di ottimizzazione a tempo di compilazione: il programma può essere reso più corto a livello di numero di istruzioni oppure può essere reso più breve come tempo di esecuzione.

Il programma simulato non può ancora interagire con l’esterno (non c’è tastiera né display per l’output, né un’unità a disco per memorizzare i risultati). Inoltre non sono implementate le interruzioni (motivo per cui non ci sono ancora i dispositivi I/O):