Verslag Microprocessoren

Mathy Vanhoef

5 februari 2012

1 Structuur

De "onderste laag" bestaat uit de header file hardware.h waarin de binaire waarde van alle instructies staat, de adresseringsmethoden en het type "geheugenCell of MemCell" is gedeclareerd. Hiernaast is er ook een gelinkte lijst van MemCell structs gemaakt die het geheugen voorstelt (memory.h). Bovenop dit staat de "processor" die dit geheugen en hardware gebruikt en hiermee instructies kan gaan uitvoeren.

De "runtime" bevat dan deze processor en bied eenvoudige functies aan om een programma te laden/compileren, het programma volledig uit te voeren (tot er een fout optreed of men aan een breakpoint komt), stap voor stap het programma laten uitvoeren, debuggen, enz. De runtime gebruikt hiervoor de "compiler" om een programma te laden en om te zetten naar zijn binaire code. De compiler gebruikt hiervoor dan weer gebruik van parser.h die instructies van tekst naar binair kan omzetten en omgekeerd.

Deze runtime is op zijn beurt vervat in interface.c. Deze interface is een soort commandline omgeving waarin je de uitvoering van het programma kan controleren. Door deze opbouw is het eenvoudig om eventueel een volledige grafische applicatie te maken, want om een assembler programma volledig te kunnen controleren moet je alleen de functies uit runtime.h gebruiken! Ook is het mogelijk om de output die normaal in de console verschijnt, te laten tonen in een soort text box (er wordt een pointer naar een output functie meegeven aan de runtime).

Verder is er ook nog de editor die zorgt voor een eenvoudige grafische omgeving om assembler programma's te schrijven.

2 Datastructuren

Het datatype van het geheugen is de voorgestelde union. Hierbij moest men wel letten op een detail: het lezen van een negatief getal bij onmiddellijke adressering. Omdat dit in de structure een unsigned int van 24 bits is, moet je zelf een sign extend doen om een 32 bit signed integer te bekomen.

Ook word er in het programma gebruikt gemaakt van een soort van "parse table". Dit is hier een tabel van identifiers, gelinkt aan een functie, die gebruikt wordt bij het parsen van een instructie of een commando. Zie ook sectie "Parse Functies".

3 Moeilijkheden

Echte moeilijkheden waren er niet. Alleen het maken van een goede parser om tekst om te zetten in de binaire code was het uitdagends. Men kon hier een grote functie maken met vele strcmp's en else if's, maar dit vond ik geen mooie oplossing. Ook de gekozen implementatie van de stack (voor JSB en RTS) had enkele gevolgen.

3.1 Parse functies

Om een goede parser te maken, moeten we eerst de situatie goed observeren. Er vallen enkele dingen op: Elke instructie is 3 karakters lang en het enige "argument" dat een instructie kan hebben is de adresseringsmethode. Als we nu een tabel maken met de volgende 3 velden:

- Tekstvorm van de instructie (3 karakters lang).
- Toegestane adreseringsmethoden van de instructie.
- Opcode van de instructie.

Dan hebben we genoeg info om eender welke instructie te parsen en zal dit zorgen voor korte en overzichtelijke functies. Want we moeten dan nog maar 2 functies maken. De eerste functie gaat de tabel af totdat hij bij de juiste rij is (hiervoor vergelijkt hij de input die hij krijgt met de tekstvorm van de instructie in de tabel). Eens hij dit heeft gevonden, roept hij de tweede functie aan en geeft hij de toegestane adreseringsmethoden en de opcode mee aan de functie.

Deze tweede functie hoeft dan alleen de opcode, die hij van de eerste functie meekreeg, op te slaan. Verder gaat hij elke mogelijke adresseringsmethode af die de instructie kan gebruiken. Hierdoor is de code compact maar is het toch makkelijk om eventueel instructies toe te voegen. Zie ook de code in parser.c zelf.

3.2 De stack

Omdat het een Von Neumann architectuur moest zijn, worden de return adressen in hetzelfde geheugen opgeslagen waar ook het programma en de data van het programma is opgeslagen. Het gevolg hiervan kan zijn dat de stack het programma of de data kan overschrijven. Maar dit kan ook in

echte assembler programma's gebeuren! Dus een groot probleem is dit niet. De runtime heeft ook een commando om de stack pointer aan te passen. Dus als men een zeer groot programma heeft, of er zeer veel recursieve oproepen zijn, kan men altijd voordat het programma wordt gestart deze stack pointer aanpassen.

Een leuk gevolg hiervan is dat men ook aanpassingen aan de stack kan "tracen", en zo een primitieve callstack kan maken. Hiervoor kan men de commando's "stack trace on" en "stack trace off" gebruiken.

4 Extra

Er zijn enkele extra's in het programma:

- Grafische omgeving om assembler programma's te schrijven.
- Soort van commandline interface tijdens het uitvoeren van een programma, waardoor je een grote controle hebt over de uitvoering van het programma.
- Mogelijkheid om breakpoints te gebruiken (bp, bpl, bpd)
- Uitvoeren van instructies in de commandline zonder de programma teller aan te passen (exec instructie).
- Tijdens het uitvoeren van het programma een instructie te parsen en op te slaan in het geheugen (asm adres instructie).

5 Besluit

Het is een vrij krachtige Pseudo Assembler. Je kan er programma's mee uitvoeren, stap voor stap door een programma gaan, breakpoints zetten, het programma tijdens het uitvoeren zelf aanpassen (asm), instructies via de commandline uitvoeren (exec), enz. Ook is het mogelijk om de applicatie volledig grafisch te maken.