Practicum 1: Construeer je eigen microprocessor

Processen en processoren

19 april 2011

1 Inleiding

Doel van deze opdracht is het een werkende, eenvoudige microprocessor te maken volgens de specificaties in deze tekst.

Concreet maak je de volgende onderdelen:

1. een verslag (max. 5 pagina's) van het werk dat jullie gedaan hebben, liefst een .pdf-bestand. Leg in je verslag uit welke keuzes jullie gemaakt hebben en waarom: Wat is de grote lijn van jullie ontwerp? Hoe hebben jullie de onderdelen opgebouwd? Hoe hebben jullie ze aan elkaar geknoopt? Waarom berekenen de onderdelen het juiste resultaat?

   Mij interesseert niet zozeer welke stappen je eerst en welke je later hebt gemaakt. Ik wil uit het verslag een eerste overzicht krijgen over de processor, zodat ik niet uit de vele draadjes moet achterhalen wat er is gebeurd. Ook wil ik zien welke punten die in de opdrachtomschrijving open gelaten werden jullie hebben ingevuld.

2. commentaar op de voorlopige uitwerking van een andere groep, in de tijd tussen 19 april 12.00 uur en 21 april 17.00 uur.

3. de HADES-bestanden die de CPU beschrijven (als .zip of .tar.gz-bestand)

Het beste kun je de opdracht in een groep van twee studenten maken; dan hoeven jullie samen slechts één verslag en één verzameling HADES-bestanden in te leveren. Als je erop staat kun je het werk ook alleen doen.

1.1 Deadlines

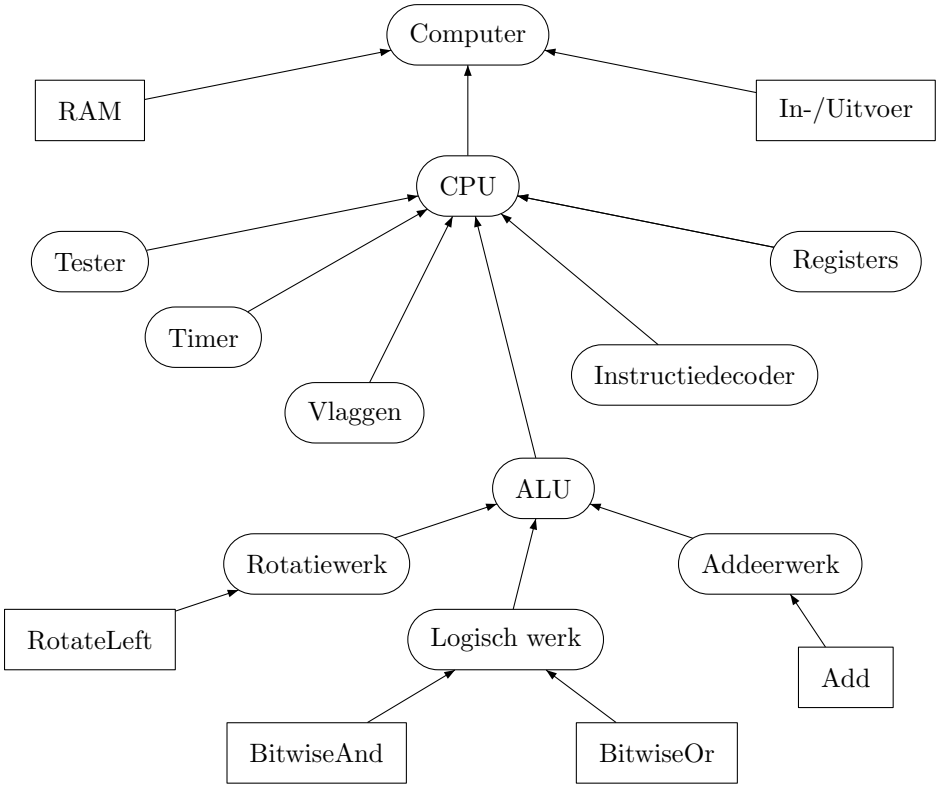
Je wordt geacht ongeveer 40 uur aan deze practicumopdracht te besteden. Om te vermijden dat je te laat begint heb ik vier deadlines gedefiniëerd, waarbij je telkens een deel moet inleveren. Als je wilt kun je vóór de eerste deadline een andere indeling met mij afspreken. Als deze deadlines je slecht uitkomen verzoek ik je z.s.m., doch uiterlijk vóór de eerste deadline hierover met mij te spreken.

25 maart 2011, 12.00 uur: Principiële opbouw van de CPU met nadruk op datastromen (Welke onderdelen heeft de CPU? Hoe moeten data door de CPU lopen om een instructie van formaat x uit te voeren? om een instructie te fetchen?), eerste deel van het verslag, één voorbeeldprogramma in assembly (liefst een programma van ca. 10–20 instructies dat één functie van de CPU test) en het rotatiewerk

1 april 2011, 12.00 uur: registerbank, ALU (compleet), vlaggen en tester

DINSDAG 19 april 2011, ~~12.00~~ 14.00 uur: toestandsmachine voor de CPU, timer die deze toestandsmachine implementeert, instructiefetcher en -decoder, voorlopige versie van het hele verslag

DONDERDAG 21 april 2011, 17.00 uur: commentaar op de voorlopige uitwerking van een andere groep.



Figuur 1: Voorgestelde structuur van de HADES-designs

vrijdag 29 april 2011, 12.00 uur: integratie van onderdelen, testen, definitief verslag

Lever alle onderdelen in bij mij, David N. Jansen, liefst per e-mail naar D.Jansen@cs.ru.nl. Stuur het commentaar op de voorlopige uitwerking op 21 april ook naar de betreffende groep.

Voor de voorlopige uitwerkingen krijg je geen cijfer, maar wel informeel commentaar. Voor overschrijding van elke deadline geldt: 1 minuut tot 2 uur te laat is $-0,25$ op je eindcijfer; 2–24 uur te laat is $-0,5$ op je eindcijfer; meer vertraging bij de eerste drie deadlines is -1 op je eindcijfer, en meer dan 24 uur vertraging bij de laatste deadline geldt als niet ingeleverd.

2 Structuur van de CPU

Een voorstel voor een structuur van de CPU wordt getoond in afbeelding 1. Het onderdeel Computer staat ter beschikking in vorm van een .hds-bestand op de website van de cursus. Van de andere ronde onderdelen moeten jullie een implementatie (een .hds-bestand) verzorgen; je mag o.a. gebruik maken van de vierkante onderdelen, die in HADES ingebouwd zijn. Je mag ook van dit voorstel afwijken, b.v. kan het zinvol zijn een functie in meerdere onderdelen op te splitsen.

3 Architectuur

3.1 Algemeen

De CPU werkt met 16 bits: registers, adressen en geheugeninhoud is telkens 16 bits breed. De CPU heeft 8 registers, $R0$ tot $R7$. Het register $R7$ is tegelijk de programmateller. Daarnaast heeft de CPU vier aparte vlaggen: negative, overflow, zero en carry.

3.2 Externe aansluitingen van de CPU

De CPU heeft de volgende externe aansluitingen:
Uitvoer:

- een 16-bit adresbus
- een 16-bit databus voor uitvoer van de CPU
- een besturingslijn $\overline{\text{WE}}$; als deze lijn 0 is wil de processor gegevens in het RAM laten opslaan. (De CPU gaat ervan uit dat bij het eerstvolgende rising edge van de klok de gegevens worden opgeslagen.)
- een besturingslijn $\overline{\text{RE}}$; als deze lijn 0 is wil de processor gegevens uit het RAM lezen. (De CPU gaat ervan uit dat het RAM bij het eerstvolgende rising edge van de klok de gegevens op de databus heeft gezet.)
- een besturingslijn HALTED; als deze lijn 1 is, is de processor gestopt.

Invoer:

- een 16-bit databus voor invoer in de CPU
- een besturingslijn $\overline{\text{RESET}}$: als deze lijn 0 is, gaat de processor zo snel mogelijk in de begintoestand; zodra de lijn op 1 gaat, begint hij te werken.
- een besturingslijn CLK die klokpulsen aan de CPU geeft. Let erop dat je geen andere klokken in de CPU inbouwt.

We hoeven de stroomtoevoer in HADES niet te modelleren.

3.3 Geheugen en in-/uitvoer

De CPU kan 65.536 verschillende adressen genereren; op elk adres wordt een woord van 16 bits opgeslagen[1]. De adressen FF00_{hex}–FFFF_{hex} dienen voor in- en uitvoer, onafhankelijk ervan welk adres in dat bereik gekozen wordt. Als de CPU iets in die adressen opslaat, worden de laagste 8 bit van het opgeslagene als ASCII-code geïnterpreteerd en op het scherm getoond. Als de CPU iets van die adressen leest, krijgt hij FFFF_{hex} als de gebruiker sinds de laatste lees-operatie geen toets gedrukt heeft, en anders de ASCII-code van de gedrukte toets.

3.4 Begintoestand

Als de computer start, worden alle registers en de vlaggen op 0 gezet, ook de programmateller. Dat betekent dus dat het bootstrap-programma op adres 0 opgeslagen moet worden.

4 Instructies

4.1 Instructieformaten

De meeste instructies geven in drie van de vier hoogste bits het doel-register aan: 000 betekent $R0$, 001 betekent $R1$, ..., 111 betekent $R7$. Veel instructies hebben als bron óf een register, óf een constante in twee-complement (= signed). Bit 5 geeft aan welke van de twee dat is. Als de bron een register is, geeft bit 4 (not) aan of het register normaal of geïnverteerd gelezen moet worden.

Het is de programmeur verboden andere machinecodes te gebruiken dan degene die hieronder gedefiniëerd zijn; als hij het toch doet is het gevolg onbepaald. Dat kan jullie helpen om de eenvoudigste manier te vinden de operaties te implementeren.

[1] Dit verschilt van wat gebruikelijk is; de meeste CPUs verlangen dat elk byte zijn eigen adres heeft.

formaat \ bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	0	0	1	neg	ovf	0	zro	cry
2a	0	destination			0	0	1	0	opcode		0	not	0	source		
2b	0	destination			0	0	1	0	opcode		1	signed constant				
3a	0	register			0	1	R/$\overline{\text{W}}$	0	0	0	0	not	0	addr reg		
3b	0	register			0	1	R/$\overline{\text{W}}$	0	0	0	1	signed address				
4a	0	destination			1	condition			A/$\overline{\text{M}}$		0	not	0	source		
4b	0	destination			1	condition			A/$\overline{\text{M}}$		1	signed constant				
5	1	destination			constant											

4.2 Instructies zonder parameters

formaat \ bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

In formaat 0 is er slechts één instructie:

instructie	operatie
HALT	de CPU stopt en doet niets meer tot men haar reset.

4.3 Vlaggen direct veranderen

formaat \ bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	1	0	0	1	neg	ovf	0	zro	cry

In formaat 1 is op dit moment één instructie gedefiniëerd; uitgebreidere versies van de processor zouden in dit formaat meer instructies kunnen toevoegen.

instructie		operatie
LOADFLAG	N O Z C *(of een deel ervan)*	zet de aangegeven vlaggen op 1 en de andere op 0

4.4 Rekenoperaties

formaat \ bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
2a	0	destination			0	0	1	0	opcode		0	not	0	source		
2b	0	destination			0	0	1	0	opcode		1	signed constant				

Rekenoperaties lezen het destination-register en het source-register of de constante, voeren een berekening erop uit en slaan het resultaat weer in het destination-register op; bovendien zetten ze de vlaggen: ADDC zet alle vlaggen afhankelijk van de gebruikelijke voorwaarden. AND en OR wissen de carry- en overflow-vlag altijd. ROL zet de overflow-vlag als de hoogste bit van het destination-register verandert en zet carry-vlag op de laagste bit van het resultaat. Alle rekenoperaties zetten de negative- en zero-vlag afhankelijk van het resultaat.

opcode	instructie		rekenoperatie
00	OR	dest, source	dest | source → dest
01	ADDC	dest, source	dest + source + carry-vlag → dest
10	AND	dest, source	dest & source → dest
11	ROL	dest, source	dest ROL (source & 15) → dest

4.5 Geheugenoperaties

formaat \ bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
3a	0	register			0	1	R/$\overline{\text{W}}$	0	0	0	0	not	0	addr reg		
3b	0	register			0	1	R/$\overline{\text{W}}$	0	0	0	1	signed address				

Formaat 3a en 3b dienen ertoe gegevens uit het geheugen (RAM) te lezen of erin te schrijven. Het adres is óf een register óf een constante, die je tussen vierkante haken [ ] schrijft. Het veld dat bij andere instructies de destination aangeeft is hier het register dat gelezen of geschreven wordt.

$\overline{\text{R}}$/W	instructie		operatie
0	READ	register, [address]	[address] → register
1	WRITE	[address], register	register → [address]

4.6 Voorwaardelijke operaties

formaat \ bits	F	E D C	B	A 9 8 7	6	5	4	3 2 1 0
4a	0	destination	1	condition	A/$\overline{\text{M}}$	0	not	0 source
4b	0	destination	1	condition	A/$\overline{\text{M}}$	1	signed constant	

In formaat 4 zijn er twee voorwaardelijke instructies. Deze kunnen gebruikt worden om, afhankelijk van de vlaggen, een bepaalde operatie uit te voeren. Vaak zal dat een sprong-instructie zijn (een instructie die de programmateller verandert), om ervoor te zorgen dat bepaalde delen van het programma alleen uitgevoerd worden als een bepaalde voorwaarde waar is.

A/$\overline{\text{M}}$	instructie		operatie
0	MOVIFcc	dest, source	Als de voorwaarde waar is: source → dest
1	ADDIFcc	dest, source	Als de voorwaarde waar is: dest + source → dest

Hierbij moet je „cc" vervangen door één van de onderstaande voorwaarden. (De meest linker kolom geeft het bitpatroon aan dat in de instructie komt te staan.)

condition	cc	voorwaarde	vlaggen
0000	T	altijd (true)	$0 = 0$
0001	F	nooit (false)	$1 = 0$
0010	GEU C	$\geq$ (unsigned) of carry	$0 = \neg C$
0011	LU NC	$<$ (unsigned) of niet carry	$1 = \neg C$
0100	GE	$\geq$ (signed)	$0 = N \; xor \; O$
0101	L	$<$ (signed)	$1 = N \; xor \; O$
0110	NN	≥ 0	$0 = N$
0111	N	< 0	$1 = N$
1000	NZ	$\neq 0$	$0 = Z$
1001	Z	$= 0$	$1 = Z$
1010	GU	$>$ (unsigned)	$0 = Z \vee \neg C$
1011	LEU	$\leq$ (unsigned)	$1 = Z \vee \neg C$
1100	G	$>$ (signed)	$0 = Z \vee (N \; xor \; O)$
1101	LE	$\leq$ (signed)	$1 = Z \vee (N \; xor \; O)$
1110	NO	geen overflow	$0 = O$
1111	O	overflow	$1 = O$

De operaties laten de vlaggen onveranderd, je kunt dus eventueel meerdere voorwaardelijke instructies achter elkaar plaatsen die dezelfde voorwaarde hebben.

4.7 Constante in register opslaan

formaat \ bits	F	E D C	B A 9 8 7 6 5 4 3 2 1 0
5	1	destination	constant

Omdat een register net zo groot is als een instructie is het onmogelijk een instructie te construeren waarmee je een willekeurig register op een willekeurige waarde kunt zetten.

De instructie LOADHI (formaat 5) is zo ontworpen dat je met maximaal twee instructies een register op een willekeurige waarde kunt zetten. LOADHI zet de hoogste 12 bits van de aangegeven

destination op de constante. De laagste 4 bits worden op 0 gezet. Daarna moet je eventueel OR of ADDIFT gebruiken om de laagste bits op de juiste waarde te zetten.

instructie	operatie
LOADHI dest, constant	constant $\cdot 16 \rightarrow$ dest

5 Onderdelen

Hieronder geef ik een paar hints voor de implementatie van de belangrijkste onderdelen van de CPU.

Maak vooral gebruik van de genoemde onderdelen van de RTLIB, de library van voorbereide componenten die HADES aanbiedt. Je kunt die library bekijken met het menupunt "Edit > Open component browser". Klik daarna op "built-in > hades > models > rtlib".

5.1 ALU

5.1.1 Arithmetische en logische operaties

Bruikbare bouwstenen in de RTLIB zijn b.v. rtlib.arith.Add voor een addeerwerk; rtlib.logic.BitwiseAnd en rtlib.logic.BitwiseOr voor de logische operaties. Daarmee kun je een eenheid opbouwen die de arithmetische en logische operaties direct aankan.

Invoer: 2 $\times$ 16-bit vector data; carry; lijnen om aan te geven welke operatie gewenst is: de "opcode"s 00–11.

Uitvoer: 16-bit vector data; vlaggen.

5.1.2 Rotatiewerk

De RTLIB-component rtlib.arith.RotateLeft kunnen jullie gebruiken om een vast bedrag naar links te roteren. Stel uit rotatiewerken om 1, 2, 4 en 8 bits en wat logica eromheen een rotatiewerk samen dat om elk bedrag tussen 0 en 15 bits kan roteren.

Invoer: 16-bit vector data; 4-bit vector die aangeeft hoeveel bits je moet roteren.

Uitvoer: 16-bit vector data; vlaggen.

5.2 Registers en vlaggen

De RTLIB bevat een component rtlib.memory.RegBank, maar de registers daarin worden niet goed gereset. Bouw een registerbank op uit rtlib.register.RegRE. Bedenk, afhankelijk van je datastromen, hoeveel registers je tegelijk wilt lezen en/of schrijven.

Gebruik voor de vlaggen losse flipflops. Alleen rekenoperaties en LOADFLAG veranderen hun waarde; bij andere operaties blijven de waarden van de vlaggen opgeslagen.

5.3 Tester

Voor de tester heb ik geen passende RTLIB-component gevonden; bouw hem zelf. Let op de systematische opbouw van het "condition"-veld: met de drie hoogste bits kies je een voorwaarde, met de laagste bit geef je aan of die voorwaarde 1 of 0 moet zijn. Als het hoogste bit = 1 is, is de berekening meestal $Z \vee$ (berekening als de hoogste bit = 0 is).

Invoer: "condition"-veld en vlaggen.

Uitvoer: 1 bit dat aangeeft of de voorwaarde vervuld is of niet.

5.4 Interne bussen en instructie-decodeerder

Maak eerst een principeschema van de CPU waarin jullie de routes verzamelen die gegevens in de CPU kunnen nemen: van een register naar de ALU (2$\times$), van een constante naar de ALU, van de ALU naar een register, van de vlaggen naar de tester etc. Begin bij de belangrijkste, maar ga

door tot je alle mogelijke routes hebt gevonden: kunnen alle instructies uitgevoerd worden door de routes op een geschikte manier aan of uit te zetten? Elke route wordt een (deel van een) interne bus. Daarna kunnen jullie interne besturingslijnen toevoegen die van de instructie-decodeerder naar de punten gaan waar een bus aan- of uitgezet moet worden. Om een bus uit te zetten kunnen jullie b.v. rtlib.logic.N1And gebruiken.

Jullie bouwen de instructie-decoder als "random logic". Dat betekent niet dat er iets toevalligs in zit, maar dat de decoder op een plaatje minder regelmatig eruit ziet als b.v. een stuk geheugen.

5.5 Timer

Ik stel voor dat jullie een CPU maken die zonder pipelining werkt. Dat betekent dat zij na elkaar, niet tegelijk instructies leest, decodeert en uitvoert. Maak een CPU met de volgende twee cycli:

1. een instructie *lezen:* het adres in de programmateller wordt op de adresbus gezet, de machinecode die binnenkomt wordt in een intern register van de instructie-decodeerder opgeslagen. Tegelijk berekent de CPU programmateller + 1; aan het begin van de volgende cyclus wordt die waarde in de programmateller teruggeschreven.

2. een instructie *decoderen en uitvoeren:* de decodeerder beslist wat er moet gebeuren, geeft de juiste signalen door aan de diverse onderdelen van de processor en wacht tot de uitvoerlijnen van b.v. de ALU zijn gestabiliseerd. Registers die veranderd worden worden exact aan het begin van de volgende cyclus opgeslagen.

De timer zorgt ervoor dat de andere onderdelen telkens in de juiste cyclus hun werk doen. De CPU heeft bovendien nog een paar bijzondere toestanden, die je het beste ook door de timer laat afhandelen:

Reset: Zolang de $\overline{\text{RESET}}$-invoer op 0 staat, doet de processor niets. Als de invoer naar 1 gaat, (wacht de processor even op een gunstig moment en) leest de eerste instructie.

Halted: Als de processor een HALT-instructie heeft uitgevoerd, doet hij ook niets. Hij let alleen nog op de $\overline{\text{RESET}}$-invoer.

Al deze CPU-toestanden kun je als een finite state machine modelleren en nummeren. Sla het nummer van de actuele toestand op in een paar flipflops of latches.

6 Samenvoegen en testen

Op de website staat (binnenkort) een HADES-design waar alle externe aansluitingen van de processor voorbereid en het RAM en de console al aangesloten zijn. Jullie kunnen het subdesign van de CPU daarin plakken en de verbindingen met autoconnect laten aanmaken; dan moet de computer werken.

Om te testen, stel ik ook een eenvoudige assembler ter beschikking. Jullie zullen in de komende weken ook kennis maken met een paar principes van assembly-programmering, zodat je eenvoudige programma's kunt schrijven.