Register Transfer Level systemen en de Delta I processor

Arjan J.C. van Gemund (bewerkt door Arjan van Genderen)

Dit diktaatje beoogt een bekopte aanvulling te geven bij de sheets van hoorcollege 10. In Hoofdstuk 1 wordt uitgelegd wat Register Transfer Level (RTL) systemen zijn. In Hoofdstuk 2 wordt een kenmerkend voorbeeld van RTL systemen besproken, namelijk de Delta I processor.

1. RTL systemen

Register Transfer Level (RTL) systemen zijn digitale systemen waarbij de gebruikte bouwblokken van *register* nivo (register level) zijn (dus geen losse gates maar complete, bv. 8-bits brede data registers). In het navolgende wordt uitgelegd hoe men tot deze standaardvorm van systeemopbouw komt.

1.1. Finite State Machines

De FSM is een goed voorbeeld van hoe binnen een digitaal systeem onderscheid kan worden gemaakt tussen typen subsystemen. Zoals eerder behandeld bestaat een FSM in het algemeen uit

- een register subsysteem (aantal D-FFs) tbv. het onthouden van de huidige state, en
- twee combinatorische subsystemen, respectievelijk tbv. het uitrekenen van de nieuwe state op basis van inputsignalen en huidige state, en tbv. het uitrekenen van de outputsignalen behorende bij de huidige state.

Een voorbeeld is de FSM die de paritychecker implementeert die behandeld is in hoorcollege 4 en ook getoond wordt in 10 (slide 4)

De filosofie achter deze standaardopdeling in subsystemen is dat dit het *ontwerp* van digitale systemen waar state een rol speelt (en de overgrote meerderheid van applicaties *zijn* nu eenmaal sequentieel) *vergemakelijkt*. De procedure hoe men tot het ontwerp van deze subsystemen komt is reeds behandeld in met name hoorcollege 4. De bouwblokken waarmee de subsystemen worden gerealiseerd zijn gates en D-FFs. Bij kleine digitale systemen is dit elementaire nivo van opdeling in *gate-level* combinatoriek + D-FFs toereikend. Voor het systematisch ontwerpen van complexere systemen is echter een hoger nivo van opdeling benodigd, namelijk het zogenaamde *datapadbesturingsmodel*.

1.2. Datapad-besturingsmodel

In het datapad-besturingsmodel wordt een digitaal systeem onderverdeeld in

- het datapad, een subsysteem dat de gewenste berekeningen tussen input en output verzorgt, en
- de besturing, een subsysteem dat het datapad zodanig bestuurt dat het datapad de gewenste deelberekeningen op het juiste moment uitvoert.

Een simpel voorbeeld is de reactiemeter die in behandeld wordt in hoorcollege 10 (zie slide 5). De besturing bestaat (zoals gebruikelijk) uit een FSM die – afhankelijk van signalen van buiten (zoals `start`/`stop`), en signalen vanuit het datapad (zoals `d_rdy`) – de werking van het datapad bestuurt (mbv. signalen zoals `t_clr`). Merk op dat ook het datapad kan bestaan uit meerdere FSM's (elke counter is gebaseerd op een FSM!). Met andere woorden, er is in het algemeen een hierarchie, waarbij een complex digitaal systeem bestaat uit minstens één besturingsubsysteem en één datapad-subsysteem, die zelf weer uit diverse FSM's kunnen bestaan. Natuurlijk kan een zeer complex digitaal systeem weer zijn opgebouwd uit complexe deelsystemen, en dus uit diverse besturingen en datapaden bestaan. Het datapad-besturingsmodel is een zeer handig concept gebleken om de complexiteit van het ontwerpen van grotere digitale systemen in de hand te houden.

1.3. Register Transfer Level systemen

Zoals eerder uitgelegd, bestaat het datapad bij grotere digitale systemen zoals de reactiemeter niet uit gate-level componenten maar uit complexere bouwblokken zoals complete (n-bits) tellers, decoders,

etc. Bij een belangrijke klasse van complexe digitale systemen, *register transfer level* systemen genoemd, bestaat het datapad voor een belangrijk gedeelte uit *registers*. Bij dit soort systemen moet het datapad diverse berekeningen op data (gegevens) uitvoeren waarbij er diverse deelstappen in het berekeningsproces zijn te onderscheiden. Vaak is het nodig om per deelstap de data tijdelijk te kunnen bewaren in registers zodat de data bij een latere deelstap beschikbaar is. Bij RTL systemen wordt in het datapad onderscheid gemaakt tussen

- functionele bouwblokken, subsystemen die gegevens op de ingang(en) volgens een of andere functie verwerkt en het resultaat aanbiedt op de uitgang(en), en
- registers, om (deel)resultaten te kunnen vasthouden.

De term "register transfer level" geeft niet alleen aan dat het nivo van gehanteerde bouwblokken ligt op het nivo van registers, maar geeft ook aan dat de operaties in het datapad worden beschreven op het nivo van gegevensuitwisseling tussen registers (register transfer). Een voorbeeld van zo'n RTL systeem is de implementatie van het gcd-systeem zoals afgebeeld in hoorcollege 10 (slide 6). Het gcd-systeem is een digitaal systeem dat de grootste gemene deler (greatest common divisor, gcd) van de positieve integers x en y berekent volgens het onderstaande algoritme, waarbij een hulpfunctie *mod* (modulo) en drie variabelen x, y, en z zijn betrokken:

gcd (x,y):

if (y = 0) **then**
klaar: gcd is x
else
z := mod(x,y)
x := y
y := z
repeat gcd(x,y)

Bijvoorbeeld voor de ingangswaarden x = 24 en y = 15 volgt (telkens op het moment dat de **if** wordt uitgevoerd):

stap:	0	1	2	3	4	
x:	24	15	9	6	3	
y:	15	9	6	3	0	→ klaar: gcd = 3
z:	---	9	6	3	0	

Het datapad van het gcd-systeem bestaat uit 3 (n-bits) registers die de variabelen x, y, en z realiseren, en een combinatorisch subsysteem die de modulo functie realiseert. Het datapad, dat de data-bewerkingen van bovenstaand algoritme implementeert (y = 0, z := mod(x,y), x := y, y := z), staat onder besturing van een (simpele) FSM die de zogenaamde *control flow* van bovenstaand algoritme implementeert (**if** .. **then** .. **else**, **repeat**). De test (y = 0) is gerealiseerd mbv. een NOR gate en vormt de terugkoppeling die bepaalt of de FSM kan stoppen of dat de FSM de register transfer operaties z := mod(x,y), x := y en y := z dient uit te voeren. In de in dit voorbeeld gekozen realisatie (er zijn ook snellere oplossingen mogelijk, zoals we later zullen zien) wordt op elke actieve klokflank precies één van de drie registers geladen met een nieuwe waarde. Eerst wordt het z-register (vanaf nu z genoemd) geladen met de uitgangswaarde van de mod combinatoriek (mbv. het synchrone besturingssignaal `ld_z` is 1 gemaakt na de voorafgaande actieve flank). Op de volgende actieve klokflank wordt x overschreven (mbv. `ld_x`) en op de volgende actieve klokflank wordt y overschreven (NB: de volgorde van overschrijven is *niet* willekeurig), waarna de volgende iteratie wordt ingegaan.

1.4. Processoren

Tot nu toe hebben we gezien hoe het ontwerp van digitale systemen kan worden vereenvoudigd door het gebruik van standaard concepten (FSM model en het datapad-besturingsmodel waaronder met name het RTL model). Het is echter nog wel zo dat het ontwerp van besturing en datapad geheel *specifiek* is voor de applicatie. Bijvoorbeeld zowel de FSM als datapad van de reactiemeter zijn geheel anders dan de FSM en datapad van het gcd-systeem. Dit houdt in dat – ondanks de standaardisatie mbt. de scheiding van besturing en datapad – het ontwerpen van een complex digitaal systeem zeer tijdrovend kan zijn.

Met name de introductie van het RTL model heeft een trend veroorzaakt waarbij het RTL datapad zo wordt ontworpen dat het multi-functioneel ("general-purpose") wordt en derhalve kan worden *hergebruikt* tbv. *verschillende* applicaties. Dit levert een belangrijke winst in ontwerpsnelheid, aangezien nu slechts de applicatiespecifieke besturing moet worden ontworpen. Het meest kenmerkende voorbeeld van zo'n "standaard" RTL datapad is te vinden in de klasse van digitale systemen die gewoonlijk *processoren* worden genoemd. De benaming slaat op de naam van het datapad dat formeel "data processor" wordt genoemd vanwege diens (veelzijdige) *data processing* mogelijkheden. Als voorbeeld van een RTL systeem met zo'n datapad behandelen we de Delta I processor, die ook op het prakticum (p3) zal worden (af)gebouwd en toegepast in de eindopdracht.

2. De Delta I processor

De Delta I processor is een Delfts ontwerp dat speciaal tbv. Digitale Systemen is ontworpen vanuit slechts één doelstelling: het ontwerp moet zo *simpel mogelijk* zijn. Het ontwerp moet illustreren dat een processor in principe uit slechts een handvol onderdelen is te maken. Tevens moet de scheiding tussen datapad en besturing duidelijk herkenbaar zijn. Bovendien moet de instructieset-architectuur (waarover later meer) zo simpel zijn dat deze snel te leren is, en tegelijkertijd voldoende mogelijkheden biedt om applicaties te kunnen realiseren. De vele – soms zeer uitgebreide – processorontwerpen die men in diverse leerboeken op het gebied van de Digitale Techniek kan vinden, schieten in dit didactisch opzicht helaas te kort.

De Delta I (de aanduiding "I" staat voor eerste generatie, het is niet geheel uitgesloten dat er opvolgers komen met een wat luxere architectuur) bestaat uit een 8-bits datapad dat wordt bestuurd door een FSM (zie slide 11 van hoorcollege 10). Het datapad bestaat uit een aantal registers, een ALU en een aantal tri-state buffers, allen verbonden dmv. een 8-bits databus. De besturing bestaat uit een 8x12 bit ROM, een 8-bits counter en een combinatorisch subsysteem, genaamd instructie-decoder. Een minimaal geconfigureerde Delta I processor die kan rekenen en met de buitenwereld kan communiceren bestaat uit 2 registers, 3 tri-state buffers, ALU, counter, ROM en instructiedecoder, in totaal dus 9 componenten.

2.1. Datapad

Het datapad is opgebouwd uit subsystemen die onderling zijn verbonden via de databus (zie slides hoorcollege 10). Er zijn drie typen subsystemen:

- Data-registers R_0 … R_{D-1} met bijbehorende tri-state buffers. De *D* dataregisters zijn bedoeld om (8-bit) gegevens tijdelijk op te slaan. Gegevens kunnen via de bus in het data-register worden opgeslagen dmv. de synchrone load ingang `dreg_ld`, en via de tri-state buffer dmv. de synchrone output enable ingang `dbuf_oe` op de bus ter beschikking worden gesteld. De data-registers vormen de "RAM" van de Delta I.
- I/O-registers R_D … $R_{D+...}$ met bijbehorende tri-state buffers. Deze registers zijn genummerd vanaf het laatste data-register. De I/O-registers (I/O = input/output) zijn bedoeld om gegevens uit de buitenwereld in te lezen (via de tri-state buffer) en naar de buitenwereld te sturen (via het register, merk op dat in vergelijking met het dataregister de verbinding tussen register en buffer is "opengeknipt"). Het totale aantal data- en I/O-registers is beperkt tot 256 vanwege de gekozen instructieset-architectuur (waarover later meer). Dit aantal is echter ruimschoots toereikend voor vele applicaties (waaronder de eindopdracht).
- ALU met bijbehorende accumulator A en tri-statebuffer. De accumulator, ook wel A-register genoemd, is een data-register dat samen met de ALU de rekenkern van de processor vormt. De ALU kan – afhankelijk van de aangeboden 3-bits besturingscode "ALU_code" – elementaire rekenoperaties zoals optelling, bit-wise AND, XOR, etc. uitvoeren op de 8-bits ingangsoperanden. De ene operand is afkomstig van de databus (dwz. van hetzij een data-register, een I/O-register, of de instructie-decoder) terwijl de andere operand altijd door het A-register wordt geleverd. Doordat het resultaat van de ALU op deze wijze altijd weer wordt teruggegekoppeld als operand heeft het A-register vaak een accumulerende functie. Bijvoorbeeld als de ALU meerdere optellingen achtereen uitvoert bevat het A-register steeds de totale som van alle voorgaande optellingen. Dit verklaart de naamgeving "accumulator". Wij zullen in het vervolg ook vaak de termen A en A-register gebruiken als we het over de accumulator hebben. Het opslaan van de ALU-uitgangsdata in A geschiedt dmv. het synchrone load signaal "areg_ld". De inhoud van A kan op de bus worden aangeboden via de

tri-state buffer dmv. de synchrone output enable ingang “abuf_oe”. Naast rekenkern heeft de ALU ook een belangrijke functie als terugkoppeling op de besturing. Een belangrijke vereiste van een processor is namelijk dat toekomstige operaties op het datapad afhankelijk moeten kunnen zijn van eerder verkregen resultaten. Hiertoe is de ALU uitgevoerd met 2 extra uitgangsbits, te weten de Z (“zero”) en C (“carry”) bits (ook wel *flags* of “vlaggen” genoemd). Deze bits kunnen door de ALU worden gezet als gevolg van een aritmetische operatie (bv. C := 1 als overflow optreedt tijdens een optelling, en Z : = 1 als A = 0 als gevolg van een AND-operatie). De bits zijn intern mbv. 2 D-FFs geïmplementeerd zodat ook na beëindigen van de klokcyclus de informatie behouden blijft. Hoe deze bits worden gebruikt door de instructie-decoder wordt later uitgelegd.

Het datapad staat onder besturing van de besturingsignalen `dreg_ld(0)` … `dreg_ld(D+..)`, `dbuf_oe(0)` … `dbuf_oe(D+..)`, `areg_ld`, `abuf_oe` en `ALU_code`. Op `ALU_code` na, zijn alle besturingsignalen *synchroon* hetgeen betekent dat het effect van deze signalen optreedt op de eerstvolgende actieve flank van een centrale klok `clk` (de processorklok). Door de veelheid van signaalcombinaties kan het datapad diverse operaties uitvoeren. Willekeurige voorbeelden zijn (we nemen aan dat $D = 8$):

- het *copiëren* van een extern 8-bits signaal (bv. via `input(1)`) naar A dmv. het activeren van `dbuf_oe(9)`, `areg_ld` alsmede de ALU code waarbij de ALU transparant wordt. We zullen deze operatie later conceptueel aanduiden als “het lezen van I/O register `R9`”. Het registernummer moet dus wel boven *D* liggen, anders zou de inhoud van een *data*-register in A worden gecopieerd, hetgeen overigens ook een nuttige operatie is;
- het *optellen* van A met de inhoud van data-register `R1` dmv. het activeren van `dbuf_oe(1)`, `areg_ld` en de bijbehorende ALU code (het effect is dus A := A + `R1`);
- het *copiëren* van A naar uitgang `output(1)` dmv. het activeren van `abuf_oe` en `dreg(9)`. We zullen deze operatie later aanduiden als “het schrijven naar I/O register `R9`”. Het registernummer moet weer boven *D* liggen, anders zou de inhoud van A in een *data*-register worden gecopieerd, hetgeen overigens ook een nuttige operatie is.

Het gehele repertoire van transportoperaties en aritmetische operaties wordt later behandeld.

2.2. Besturing

De eerdergenoemde besturingsignalen zijn afkomstig van de instructie-decoder. Dit is een combinatorisch subsysteem die samen met de 8-bit tri-state buffer, 8-bit counter en 8x12 bit ROM de FSM vormt die het datapad bestuurt. De counter vormt het state register van de FSM terwijl de ROM en instructie-decoder de gezamenlijke combinatoriek verzorgen tbv. de volgende state en datapad besturing. De opsplitsing tussen ROM en instructiedecoder is kenmerkend voor processorarchitecturen en biedt de mogelijkheid om de instructie-decoder geheel *applicatie-onafhankelijk* te houden. Dit houdt in dat slechts de ROM de enige component is die nog applicatiespecifiek is! Elke bitcode in de ROM – ook wel *instructie* genoemd – bepaalt welke besturingsignalen door de instructie-decoder worden geactiveerd om op de eerstvolgende actieve klokflank de gewenste operatie tot gevolg te hebben. De opbouw van de instructie-code wordt later behandeld. Het afleiden (decoderen) van de vele besturingsignalen uit de 12-bit instructiecode gebeurt in de instructie-decoder. Naast de eerder beschreven besturingsignalen van het datapad zijn - tbv. de besturing zelf - de volgende besturingsignalen van belang:

- `pc_inc`: dit besturingsignaal zorgt dat na de eerstvolgende actieve flank van de processorklok de inhoud van de counter met 1 is verhoogd (`pc_inc` is verbonden met de synchrone `increment` ingang van de upcounter). Dit heeft tot gevolg dat de de instructie-code in het volgende ROM adres aan de instructie-decoder wordt aangeboden ter decodering. De reeks ROM instructies wordt ook wel met de term *programma* aangeduid. Dit verklaart waarom de counter meestal wordt aangeduid met de term PC (program counter). Vanaf nu zal het counter-register met de term PC worden aangeduid. De meeste ROM instructies zijn zodanig dat na de uitvoering van de instructie door het datapad (op de eerst-volgende actieve klokflank) de volgende instructie moet worden ge-adresseerd.
- `pc_ld`: dit signaal biedt de mogelijkheid om de PC via de databus een geheel andere waarde te geven (`pc_ld` is verbonden met de synchrone `load` ingang van de counter). Dit nieuwe ROM adres kan worden aangeboden vanuit de instructie-decoder (dus door bitcodes vanuit de ROM zelf, via de 8-bit tri-state buffer dmv. het synchrone output enable signaal `ibuf_oe`),

of vanuit een 8-bit data-register (dmv. het output enable signaal van de bijbehorende tri-state buffer). In het eerste geval wordt het adres van de volgende instructie dus door de huidige instructie bepaald. In het tweede geval wordt dit bepaald door het datapad (bv. door de uitkomst van een *berekening*). In beide gevallen zullen we spreken van een "sprong-opdracht" oftewel *jump* of *branch* instructie (waarover later meer).

- `ibuf_oe`: naast de rol van dit besturingsignaal in het realiseren van sprong-opdrachten speelt dit signaal ook een rol met betrekking tot het datapad. Door het activeren van `ibuf_oe` in combinatie met bv. `areg_ld` kunnen bitcodes vanuit de ROM (via de instructie-decoder) worden geladen in A. Dit is een belangrijke voorziening omdat registers vaak geïnitialiseerd moeten kunnen worden met codes vanuit het ROM-programma en niet alleen vanuit de buitenwereld (via de input-lijnen).

Met bovenstaande besturingsignalen `pc_inc`, `pc_ld` en `ibuf_oe` zijn alle besturingsignalen van de Delta I behandeld.

2.3. Instructieset-architectuur

Met bovengenoemde besturingsignalen is er een veelheid aan transportoperaties en aritmetische operaties mogelijk. In principe ondersteunt het data- en besturingspad alle mogelijke combinaties van gegevenstransport tussen register R_i naar R_j, R_i naar A, R_i naar PC, ROM naar R_i, A naar PC, A naar R_i, en ROM naar A (zie slides hoorcollege 10). Echter indien alle mogelijkheden door middel van instructie-codes zou worden aangeboden zou het aantal bijbehorende codes zo groot worden dat de bitlengte van de instructie erg lang zou worden. Bijvoorbeeld de bitcode van een instructie die tot gevolg moet hebben dat R_i naar R_j zou worden gecopieerd, zou de volgende bitvelden moeten hebben: een aantal bits om het type "register-naar-register" te coderen (de *operatiecode* of *opcode* genoemd), een 8-bit code (een *operand* genoemd) om aan te geven welk bronregister R_i we bedoelen (er zijn 256 registers dus $i = 0 \ldots 255$), en nog eens een 8-bit operand om aan te geven welk doelregister R_j we bedoelen (idem dito). Dit houdt in dat we een ROM van meer dan 20 bits breed nodig hebben (aannemende dat we voor het gemak elke instructie dezelfde bitlengte geven). In de Delta I is gekozen voor een *instruction format* (instructie-opbouw, ook wel aangeduid met de term *instructieset-architectuur*) die naast de operatiecode slechts *één operand* kent. Het aantal mogelijke transportoperaties (dus de omvang van de *instructieset*) is hiermee zo laag geworden dat het gehele instructie-repertoire binnen de 16 types blijft. Derhalve zijn slechts 4 bits voor de opcode benodigd. Samen met het operandveld van 8 bits hebben we dus een totale instructielengte van 12 bits (het exacte instructieformat wordt later behandeld).

Verschillende slides van hoorcollege 10 geven een illustratie van de verschillende typen operaties die met 1 operand kunnen worden gerealiseerd. De niet bij de operatie betrokken componenten zijn gestippeld weergegeven. Doordat er slechts 1 operand kan worden opgegeven werken veel instructies met A (dat kost geen extra operand). Er zijn 3 instructiecategorieën:

- transportinstructies zoals
  - het laden van operandwaarde "#" in A, dus A := # (de "`ld A #`" instructie, "ld" staat voor "load"), zie sheet 1-13;
  - het laden van de inhoud van data-register R_i in A, dus A := R_i (de "`ld Ri`" instructie, $i < D$), sheet 1-14;
  - het laden van het inputkanaal `input(i-D)` in A (de "`ld Ri`" instructie, $i >= D$, NB: de opcode is hetzelfde als bij de vorige instructie, echter de waarde van *i* bepaalt of we met een data-register of een I/Oregister te maken hebben), sheet 1-15;
  - het bewaren van A in data-register R_i, dus R_i := A (de "`st Ri`" instructie, $i < D$, "st" staat voor "store"), sheet 1-16;
  - het "bewaren" van A in I/O-register R_i (de "`st Ri`" instructie, $i >= D$, de data is beschikbaar op het 8-bit output kanaal `output(D-i)`), sheet 1-17;
- aritmetische instructies zoals
  - het optellen van A met operandwaarde "#", dus A := A + # (de "`add #`" instructie), sheet 1-18;
  - het optellen van A met de inhoud van register R_i dus A := A + R_i (de "`add Ri`" instructie, met betrekking tot de waarde van *i* gelden dezelfde regels als eerder uitgelegd), zie sheet 1-19;
- spronginstructies zoals

- laad PC met operandwaarde "#" dus PC := # (de "`jp #`" instructie, "jp" staat voor "jump"). Merk op dat bij alle voorgaande load- store- en aritmetische instructies de PC simpelweg wordt geïncrementeerd dmv. `pc_inc`, bij spronginstructies is de situatie anders, zie sheet 1-20;
- laad PC met de inhoud van register R_i dus PC := R_i (de "`jp Ri`" instructie, met betrekking tot de waarde van *i* gelden weer dezelfde regels als eerder uitgelegd), sheet 1-21;
- laad PC met operandcode "#" dus PC := # *indien* het ALU carry bit *C* = 1, dwz. indien een eerdere aritmetische instructie een *carry* heeft opgeleverd. Als dit *niet* het geval is, heeft de instructie geen effect en geldt PC := PC + 1 (de "`br #`" instructie, "br" staat voor "branch", NB: branch is dus een *conditionele* jump), zie sheet 1-22.

Bovenstaande voorbeelden geven een goede indruk wat er met een 1-operand instructie format en dit datapad mogelijk is. Het instructie-repertoire dat de Delta I ondersteunt is gegeven in slide 22 van hoorcollege 10. Let op de invloed van `C` en `Z` op de operaties en vice versa. De waarheidstabel van de instructie-decoder is weergegeven in slide 23 van hoorcollege 10. Het 12-bit instructie format bestaat uit een 4-bit opcode veld en een 8-bit operand veld. Deze 12 bits worden (samen met de Z en C bits van de ALU) gedecodeerd tot de besturingsignalen `pc_inc`, `pc_ld`, `ibuf_oe`, `areg_ld`, `abuf_oe`, een grote hoeveelheid (`dreg_ld(i)`, `dbuf_oe(i)`) signaalparen voor de data- en I/O-registers, en de 3 ALU code bits. Met bovenstaande kennis van datapad en besturing is de waarheidstabel goed te volgen en vormt feitelijk de beste specificatie van de werking van de Delta I vanuit instructie-perspectief.

We eindigen deze paragraaf met een ROM-programma'tje dat de herhaaldelijk de berekening `R0 := R1 + 15` uitvoert (zie slide 24 hoorcollege 10; of `R0` en `R1` data- of I/O-registers zijn is in onderstaand verhaal niet relevant):

ROM adres	instructie-code	operatie	commentaar
`00000000`	`0010 00000000`	`clr c`	`C := 0, PC := PC + 1`
`00000001`	`0100 00000001`	`ld R1`	`A := R1, PC := PC + 1`
`00000010`	`0110 00001111`	`add 15`	`A := A + 15, PC := PC + 1`
`00000011`	`0101 00000000`	`st R0`	`R0 := A, PC := PC + 1`
`00000100`	`1100 00000000`	`jp 0`	`PC := 0`

Na reset (signaal: `rst`) staan alle registers (dus ook de PC) op nul, dus zal de Delta I starten met het uitvoeren van de instructie-code op ROM-adres 0. Instructie (adres) 0 zorgt ervoor dat (na de eerstvolgende actieve klokflank) de carry C van de ALU wordt gereset. Anders bestaat later het risico dat de carry wordt meegeteld in de "add" instructie waardoor er 16 bij A zou kunnen worden opgeteld ipv. 15). In instructie 1 wordt R1 in A geladen (geen effect op C). In instructie 2 wordt de constante 15 uit de ROM code aan de ALU aangeboden en bij A opgeteld, waarna het resultaat van de ALU weer in A wordt geklokt. Het is illustratief om even na te gaan hoe zo'n instructie precies werkt. De twee operanden (A en 15) staan feitelijk al een hele klokperiode (minus wat propagatietijden) klaar aan de ingang van de ALU, namelijk direct na de klokflank die er voor zorgt dat de PC op `00000010` is gezet. Op dat moment (afgezien van propagatietijden) staat de instructie `0110 00001111` op de ROM-uitgang, en wordt deze code direct (afgezien van propagatietijden) gedecodeerd door de instructie-decoder, met als gevolg dat `pc_inc` = 1, `ibuf_oe` = 1, `areg_ld` = 1 en `alu_code` = `011`. Doordat `ibuf_oe` = 1 verschijnt het getal 15 direct (afgezien van propagatietijden) op de databus (de andere ALU operand, A, staat altijd al op de andere ALU ingang). Afgezien van de propagatietijd die de ALU nodig heeft om beide operanden bij elkaar op te tellen, staat het resultaat A+15 dus direct ter beschikking op de ALU uitgang. Pas op de *volgende* actieve klokflank wordt het resultaat in A geklokt (A wordt dan dus overschreven). Tegelijkertijd incrementeert de PC counter, zodat de volgende ("`st`") instructie op de uitgang van de ROM verschijnt (waarop het datapad een nieuwe instelling krijgt, echter het vorige resultaat is dus al veilig in A geklokt). Het executeren van een instructie kent dus twee fasen: de fase na de huidige actieve flank waarop het datapad zich instelt op de huidige operatie (en die deels uitvoert, zoals de ALU), en het moment van de volgende actieve flank, waarop de betrokken registers (waaronder altijd de PC) de verkregen waarden inklokken, waarop het datapad zich alweer instelt op de nieuwe operatie. We keren terug naar het voorbeeld-programma. De berekening R1 + 15 had overigens ook kunnen worden gerealiseerd dmv. "`ld 15`"- en "`add R1`"-instructies. In instructie 3 wordt het resultaat in A weggeschreven naar R0. Met het C bit dat eventueel

is ontstaan na de "`add`" instructie wordt verder niets gedaan. In instructie 4 (dus eigenlijk "op de actieve klokflank volgend op instructie 3") wordt de PC van een nieuwe waarde (0) voorzien, zodat de instructies opnieuw worden doorlopen. Omdat het onzeker is of naar aanleiding van de vorige "`add`"-insteructie C = 1 is geworden, is de eerste instructie bepaald niet overbodig (de waarde van C wordt namelijke niet beïnvloed door "`st`" en "`jp`" instructies; dit is een bewuste ontwerpkeuze omdat het vaak handig is om deze informatie een aantal instructies verderop nog ter beschikking te hebben, hier is het toevallig zo dat deze architectuur-keuze een extra instructie kost).

Omdat het programmeren van instructie-bitcodes erg foutgevoelig is, geeft men er de voorkeur aan om programma's eerst te ontwerpen in termen van omschrijvingen zoals "`ld R1`" die nu eenmaal makkelijker zijn te onthouden en te lezen dan moeilijke bitcodes. Zo'n operatie-omschrijving wordt *mnemonic* genoemd. Een tweede reden om programma's eerst in termen van mnemonics op te schrijven is dat dit een zekere "portabiliteit" geeft: mocht er ergens een instructie tussen moeten, dan hoeven niet allerlei adressen te worden veranderd. Een aanvullend mechanisme dat tbv. van programmeergemak is ingevoerd is het gebruik van *labels*. Het is handiger om ipv. het adres 0 in "`jp 0`" een label te gebruiken zoals het label "`begin`" in "`jp begin`". Het label "`begin`" verwijst naar het adres waar naar moet worden gesprongen. Mocht nu het gehele programma onverhoopt nog een stuk verplaatst moeten worden in de adresruimte, dan hoeft de programma-text niet te worden aangepast. De bovenstaande ROM code wordt derhalve eerst opgeschreven in de volgende vorm (zie slide 24 hoorcollege 10):

```
Instructie      commentaar

begin:      clr c       ; bereid optelling voor
            ld R1       ; R0 := R1 + 15
            add 15
            st R0
            jp begin    ; herhaal programma
```

Merk op dat het label waar "`jp`" naar verwijst, vóór de instructie is terug te vinden waar de processor naar toe moet springen (dwz. het adres van de instructie die de processor vervolgens moet uitvoeren).
Als het programma-ontwerp voltooid is (meestal na veel herschrijven) heeft het pas zin om de bijbehorende bitcodes en adressen te bepalen. Dit gebeurt meestal mbv. een *assembler*, een computerprogramma dat bovenstaande mnemonics en labels automatisch transformeert naar ROM-adressen en -instructiecodes. Bovenstaande reeks mnemonics en labels wordt vaak ook *assembly (source) code* genoemd. In het geval van ons voorbeeld-programma zou een Delta I assembler de reeds eerder getoonde bitcodes produceren (het label "begin" zou met adres 0 geassocieerd worden en overal in het programma waar het label "begin" als operand gebruikt zou worden, zou het operandveld `00000000` worden gegenereerd). Door middel van de ";"-characters in het commentaarveld is in bovenstaande source code al rekening gehouden met verwerking door een assembler. Meeste assemblers herkennen het ";"-character als het begin van programmeur-commentaar en slaan dan de rest van de regel gewoon over.

2.6. Applicatie-ontwerp

Zoals eerder uitgelegd, is de ROM de enige component die applicatie-specifiek is. Dit levert uiteraard een groot voordeel op met betrekking tot de snelheid en de flexibiliteit tijdens het ontwerpen van het digitale systeem. Met name bij zeer complexe applicaties zoals bijvoorbeeld een zelf-denkende robot pleegt men vrijwel uitsluitend in termen van processor-gebaseerde ontwerpen te denken aangezien het traject van algoritme (applicatie) naar geheel applicatie-specifieke digitale hardware menselijkerwijs nog voor onmogelijk wordt gehouden (hierover later meer). Ondanks het feit dat de ROM formeel gewoon een combinatorische component binnen de processor-FSM is, plegen we de waarheidstabel van de ROM veel meer te zien als een tabel van instructies (een programma), dankzij de invoering van eerder behandelde instructieset-architectuur die door het gebruik van een instructie-decoder mogelijk is geworden. Gezien het feit dat het ROM-programma feitelijk de gehele (applicatie-)functionaliteit van de hardware bepaalt maakt men onderscheid tussen de ROM *hardware* en de *inhoud* (dus de *informatie*) hetgeen men *software* noemt (makkelijker te veranderen dan de hardware dus soft). Als voorbeeld hoe men de Delta I "instantieert" tot een digitaal systeem dat een specifieke applicatie implementeert, behandelen we het gcd-systeem dat al eerder aan bod is gekomen in Hoofdstuk 1.

De Delta I implementatie van het gcd-systeem is gebaseerd op het algoritme dat eerder is behandeld. Centraal staan ook hier weer de variabelen x, y, en z die nu geïmplementeerd worden mbv. data-registers `R0`, `R1` en `R2`. Voorts hebben we nog een data-register `R3` nodig, naar later zal blijken, waarmee het aantal benodigde data-registers $D = 4$ bedraagt. De oorspronkelijke waarden van x en y lezen we in vanuit de buitenwereld mbv. I/O-registers `R4` en `R5` (feitelijk gebeurt dat mbv. de betreffende tri-state buffers maar conceptueel en dus ook software-matig "lezen we van registers `R4` en `R5`" mbv. "`ld R4`" en "`ld R5`"). Het resultaat $z = \gcd(x,y)$ schrijven we naar de buitenwereld via register `R6` (nu gebruiken we *wel* een echt register). Merk op dat we ook `R4` of `R5` hadden kunnen "hergebruiken": immers "`ld Ri`" gebruikt tri-state *buffer i* terwijl "`st Ri`" *register i* gebruikt. Gezien de hoeveelheid registers die ter beschikking staan (256) kiezen we er echter voor om een nieuwe index te kiezen om de kans op programmeerfouten te verkleinen. De oorspronkelijke invoerwaarden x en y moeten dus via de 8-bits inputkanalen `input(0)` en `input(1)` aan de Delta I worden aangeboden terwijl het resultaat z via 8-bit outputkanaal `output(2)` beschikbaar komt.

De assembly code ziet er als volgt uit (met inbegrip van commentaar-regels die goed gebruik zijn bij de ontwikkeling van – ook voor andere programmeurs begrijpelijke – code, zie ook slides 25-27 hoorcollege 10):

```
; Bereken gcd met de Delta I
;
; Delta I configuratie:
;
; D = 4 data-reg: R0 (x)
;       R1 (y)
;   R2 (z)
;   R3 (return adres)
; 3 I/O reg: R4 (x)
;   R5 (y)
;   R6 (gcd(x,y))
;
;
; Algoritme:
;
;  als (y = 0) dan
;  klaar: x is gcd
;  anders
;  z := x mod y
;  x := y
;  y := z
;  herhaal algoritme

gcd:  ld R4       ; x := input(0)
      st R0
```

```
      ld R5       ; y := input(1)
      st R1
loop: ld R1       ; test y = 0
      and 11111111b
      bz done     ; y = 0 dus we zijn klaar
      ld after    ; R3 := terukeeradres na executie
      st R3       ; van de mod procedure
      jp mod      ; roep mod aan => z := mod(x,y)
after:ld R1       ; x := y
      st R0
      ld R2       ; y := z
      st R1
      jp loop     ; herhaal algoritme
done: ld R0       ; output(2) := x
      st R6
      jp gcd      ; doe een nieuwe gcd

; Bereken z := x mod y
;
; Algoritme:
;
;  z := x
; herhaal
;  z := z - y
; totdat z - y < 0
;  z := z + y

mod:  ld R0             ; z := x
      st R2
subtr:ld R1             ; z := z - y
      xor 11111111b     ; A := y'
      set c             ; A := y' + 1 = -y
      add R2            ; A := z - y
      st R2             ; z := z - y
      and 10000000b     ; z - y < 0 ?
      bz subtr          ; nee, herhaal subtr
      ld R2             ; ja, z := z + y
      clr c             ; zorg voor geen verassingen
      add R1
      st R2
      jp R3             ; return naar adres in R3
```

Er zijn bij dit programma een paar opmerkingen te maken. De "`and 11111111`" operatie is een manier om te testen of A = 0. Alleen in dàt geval levert de bit-wise AND een `00000000`-resultaat op en wordt in dat geval de Z bit (Z-vlag) op 1 gezet. De volgende branch instructie springt ("branches" in het Engels) in dat geval naar het adres in de operand (in dit geval het adres van de "`ld R0`" instructie aan het einde van het gcd-programma). Anders heeft de branch instructie geen effect en wordt de "`ld after`" instructie uitgevoerd. Dit instructietype "`ld #`" laadt het (8-bit) adres dat behoort bij het label "`after`" (het ROM adres waar de "`ld R1`" instructie staat) in A. Vervolgens wordt het adres in `R3` bewaard voor later. Deze taktiek hoort bij het gebruik van subroutines. In het gcd-algoritme wordt gebruik gemaakt van de modulo functie (zoals reeds in Hoofdstuk 1 is besproken). In plaats van de bijbehorende code gewoon in te voegen in de gcd-code, is hier gekozen om de modulo berekening in de vorm van een aparte subroutine (beginnend vanaf het adres van het "`mod`" label) te coderen, zodat deze code ook voor andere programma's gebruikt kan worden. Het mechanisme van aanroep ("call") en terugkeer ("return") wordt in de Delta I gerealiseerd mbv. een data-register (een van de "trade-offs" van de simpele Delta architectuur is dat er geen stackmechanisme ondersteund wordt). Met behulp van de instructie "`jp mod`" wordt naar de modulo subroutine gesprongen. Nadat de subroutine beëindigd is wordt er mbv. de "jp R3" instructie weer teruggesprongen naar de instructie die volgt op de "`jp mod`" instructie. Dit mechanisme, waarbij een subroutine terugspringt naar het adres in een – speciaal daarvoor gereserveerd register – maakt het mogelijk om vanuit vele punten in een programma gebruik te maken van de modulo-"service" die de subroutine biedt. Binnen de modulo-routine wordt gebruik gemaakt van de 2's-complement representatie om het (positieve) getal *y* van *z* af te trekken (dit heeft

overigens wel consequenties voor de grenzen voor *x* en *y*!). Met "`xor 11111111`" wordt de 1's-complement van A (*y*) bepaald. Mbv. "`set c`" wordt bewerkstelligt dat bij "`add R2`" automatisch het getal 1 (nml. de carry C) bij wordt opgeteld, waardoor impliciet wordt bereikt dat de waarde *–y* bij A wordt opgeteld (m.a.w. A := *z – y*). Een negatief resultaat betekent dat de hoogste bit (2's-complement!) gelijk aan 1 is. De waarde van deze bit wordt in de "`and 10000000`" instructie getest (de andere bits worden *gemaskeerd*) zodat de Z vlag 1 wordt als in A de waarde `00000000` zou resulteren. In dat geval (*z – y* >= 0) wordt opnieuw naar de instructie bij het "`subtr`" label ge"branched". De resterende code bestaat verder uit bekende operaties.

2.7. Hardware versus software

Het moge duidelijk zijn dat zelfs een processor zo simpel als de Delta I in staat is om tamelijk complexe digitale systemen te realiseren, zeker als de software-ontwikkeling kan geschieden mbv. een hogere programmeertaal zoals Java, Scheme, C, C++, Pascal, etc. (aannemende dat er een compiler beschikbaar is die programma's in zo'n taal vertaalt naar Delta I instructie-codes). De winst in ontwerpsnelheid is echter wel ten koste gegaan van de snelheid van het ontwerp zelf! Vergelijk bijvoorbeeld de snelheid van de hardware oplossing van het gcd-systeem in Hoofdstuk 1 met de processor-gebaseerde software oplossing in dit hoofdstuk. Voor onderstaande vergelijking zullen we aannemen dat – vergelijkbaar met de modulo-operatie in hardware – de software routine slechts 1 klokperiode in beslag neemt. In de gepresenteerde hardware-oplossing neemt één iteratie van het gcd-algoritme 3 klokcycli in beslag (zie FSM). In bovenstaande software-oplossing kost één iteratie maar liefst 9 klokcycli als we rekenen vanaf het label "`loop`" en als we de "`ld after`", "`st R3`", en "`jp loop`" instructies vervangen denken door één fictieve "`mod x y z`" instructie. Echter het verschil in prestatie wordt nog illustratiever als we ons bedenken dat de hardware-oplossing nog veel sneller kan door gebruik te maken van het feit dat de besturingsignalen `ld_x`, `ld_y`, en `ld_z` parallel geschakeld kunnen worden (`ld_x = ld_y = ld_z = clk`, een schuifregister-achtige realisatie). In de aldus verkregen FSM kost een iteratie slechts 1 klokcyclus waardoor de prestatie-verhouding tussen hardware-oplossing en software-oplossing 1:9 bedraagt, als we voor het gemak de klokfrequentie in beide digitale systemen gelijk veronderstellen. Hier komt nog bij dat de hoeveelheid gates in de hardware-oplossing (inclusief modulo-subsysteem) aanzienlijk minder is dan de hoeveelheid gates die benodigd is voor de Delta I (laat staan bij een commerciële "off-the-shelf" processor waarin miljoenen gates zijn verwerkt…).

De bovenstaande vergelijking illustreert dat de voordelen van een software-oplossing gepaard kunnen gaan met dramatische verliezen op het gebied van hardwareprestatie en hardwarekosten. In veel gevallen kan men zich deze kosten veroorloven (gezien de prijs en snelheid van processoren) maar de ontwerper van een digitaal systeem dient zich altijd eerst te overtuigen van de noodzaak van de gekozen oplossing. Een ontwikkeling in het voordeel van de hardware-oplossing was de komst van goede VHDL (en Verilog) compilers (*synthesizers*), die in staat zijn om een VHDL-geprogrammeerd behavioral ontwerp direct in hardware te synthetiseren. In wezen combineert deze aanpak de voordelen van een software-oplossing (de behavioral manier, waarbij het gewenste algoritme direct in VHDL kan worden uitgedrukt zoals men gewend is in een hogere programmeertaal) met de voordelen van een hardware-oplossing (de VHDL synthesizer is onder bepaalde voorwaarden in staat om tamelijk efficiente hardware te genereren die vergelijkbaar is met wat hardware-ontwerpers produceren). Deze ontwerp-procedure wordt ook wel aangeduid met *silicon compilation*: men specificeert de applicatie in software (bv. VHDL of in C, Java, etc.) en de compiler genereert geen instructies tbv. een-of-andere processor maar genereert direct de applicatiespecifieke hardware (het "silicon"). Aan de fabricage van applicatiespecifieke ICs zijn natuurlijk wel hoge initieële productiekosten verbonden (fabricage van maskers etc.) en daarom loont dit alleen voor vrij grote oplagen. Met de komst van de FPGAs is een nog belangrijke tussenoplossing ontstaan, daar FPGAs als 'standaardcomponent' gekocht kunnen worden en de hardware 'applicatiespecifiek' geconfigureert kan worden. Prestaties zijn daardoor weliswaar niet zo goed als bij een applicatiespecifiek IC, maar wel beter dan bij een processor met software oplossing.

o – o – o – o – o – o – o – o – o – o – o