Så lyfter MCU och FPGA varandra

Under utveckling och i färdig produkt

En co-processorarkitektur kombinerar en mikrokontroller (MCU) med en FPGA. MCU:n ger snabb och flexibel utveckling medan FPGA:n ger prestanda och flexibilitet. Arkitekturen är dessutom en utmärkt plattform för en iterativ konstruktionsprocess och denna artikel handlar om hur en sådan kan se ut.

Först valideras algoritmer i mjukvara (C/C++) för MCU:n. Därmed kan processer, data- och signalvägar samt kritisk funktionalitet verifieras på relativt kort tid. Genom att sedan översätta algoritmerna till en FPGA, kan konstruktören dra nytta av dess hårdvaruacceleration och modularitet.

Om systemdelar blir föråldrade eller optimeringar krävs är det därefter möjligt att göra ändringar i efterskott. Det är till och med möjligt att byta ut MCU och FPGA mot andra modeller utan att gränssnitten ändras nämnvärt.

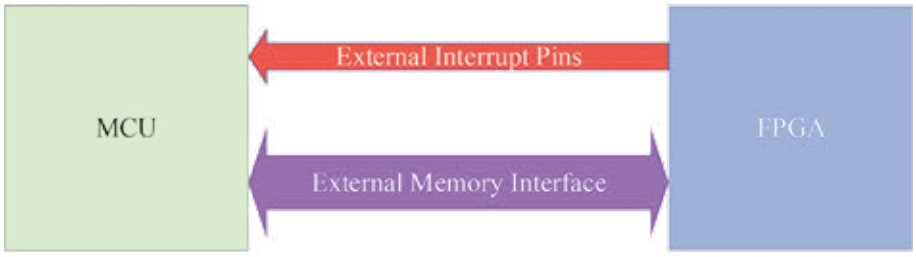
Eftersom både MCU och FPGA kan uppdateras i fält går det slutligen att göra användarspecifika ändringar och optimeringar på distans.

Följande tillämpning tjänar som exempel i denna artikel. En FPGA används – som ofta är fallet – som ett direkt gränssnitt mot en snabb AD-omvandlare. Signalen digitaliseras, läses in i FPGA:n och behandlas. Slutligen fattar FPGA:n beslut utifrån resultaten.

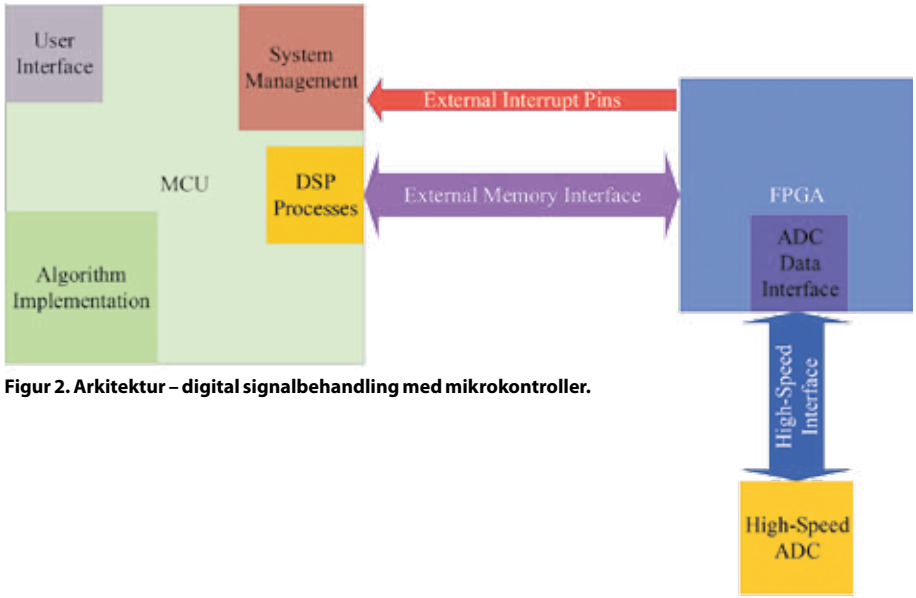
Figur 1 visar en generisk co-processorarkitektur där MCU och FPGA är anslutna via mikrokontrollerns externa minnesgränssnitt. FPGA:n hanteras som ett externt SRAM. Den skickar data tillbaka till MCU:n och fungerar som signalväg för interrupt och status. Detta betyder att FPGA:n kan indikera kritiska tillstånd för MCU:n, till exempel meddela att en AD-omvandling är klar, att ett fel uppstått eller någon annan anmärkningsvärd händelse har inträffat.

Av Rolf Horn, Digi-Key Electronics

Rolf Horn är applikationsingenjör på Digi-Key Electronics. Sedan 2014 har han ingått i företagets europeiska tekniska supportgrupp, med huvudansvar att besvara alla utvecklings- och teknikrelaterade frågor från slutkunder i EMEA. Tidigare har han arbetat på flera olika halvledarföretag med fokus på inbyggda system baserade på FPGA:er och MCU:er för industri- och fordonstillämpningar.



Figur 1. Generisk co-processorarkitektur.



Figur 2. Arkitektur – digital signalbehandling med mikrokontroller.

FÖRSTA FASEN: Digital signalbehandling med MCU

Allt annat lika går det åt mindre tid och resurser att utveckla med MCU och programvara än med FPGA och HDL-kod. Genom att starta produktutvecklingen med en MCU som huvudprocessor kan algoritmer implementeras, testas och valideras snabbare. På så sätt kan fel i algoritmer och logik upptäckas tidigt i konstruktionsprocessen, och stora delar av signalkedjan kan testas och valideras.

I denna fas är FPGA:ns roll att fungera som ett gränssnitt för snabb datainsamling. Den ska på ett tillförlitligt sätt förmedla data från en snabb AD-omvandlare, uppmärksamma MCU:n på att data finns tillgängligt och presentera data på MCU:ns externa minnesgränssnitt.

FPGA-utvecklingen i denna fas är grunden för att produkten ska bli användbar, både under själva utvecklingen och när den ska släppas på marknaden. Genom att sätta allt fokus på lågnivågränssnittet finns tillräcklig med tid för att testa de centrala operationerna.

	Latens min	Latens max	Intervall min	Intervall max	BRAM_18K	DSP48E	FF	LUT	URAM
Standard	2 935	2 935	2 935	2 935	5	1	246	964	0
Inre loop i pipeline	1 723	1 723	1 723	1 723	5	1	223	1 211	0
Yttre loop i pipeline	843	843	843	843	5	8	516	1 356	0
Arraypartitionering	477	477	477	477	3	8	862	1 879	0
Dataflöde	476	476	343	343	3	8	868	1 654	0
Inline	463	463	98	98	3	16	1 086	1 462	0

Resultat av optimeringen av FPGA-algoritmens utförande (latenstid, intervall och resursutnyttjande).

ORDLISTA

Latens: Antalet klockcykler som krävs för att utföra alla iterationer av loopen.
Intervall: Antalet klockcykler innan nästa iteration av en loop börjar bearbeta data.
BRAM: Block Random Access Memory (blockarbetsminne).
DSP48E: DSP-slice för UltraScale-arkitekturen.
FF: Flipflop.
LUT: Uppslagstabell.
URAM: Unified Random Access Memory (kan vara en enda transistor).

DE VIKTIGASTE RESULTATEN från den första fasen:

1. Hela signalkedjan – alla förstärkningar, dämpningar och omvandlingar – blir testade och validerade.
2. Projektets utvecklingstid blir minimal när algoritmer först implementeras i mjukvara. Det är av stort värde för projektledning och andra intressenter som behöver kunna bedöma om projektet är genomförbart innan de godkänner att det går vidare till konstruktionsfasen.
3. Insikter från implementeringen i C/C++ kan direkt föras över till HDL-implementeringar med hjälp av verktyg som automatisk omvandlar mjukvara till HDL, till exempel Xilinx HLS.

ANDRA FASEN: Flytta signalbehandling till FPGA

Det andra utvecklingssteget definieras av att DSP-processer och algoritmer flyttas från MCU till FPGA. FPGA:n fortsätter att vara ansvarig för det snabba ADC-gränssnittet, men genom att dessutom ge den ytterligare roller utnyttjas dess hastighet och parallellitet fullt ut. Den kan dessutom till skillnad från MCU:n köra flera instanser av signalbehandling och algoritmer parallellt.

Konstruktören kan bygga vidare på sina erfarenheter från MCU-implementeringen. Verktyg som till exempel det tidigare nämnda Vivado HLS från Xilinx kan översätta från C/C++-kod till syntetiserbar HDL.

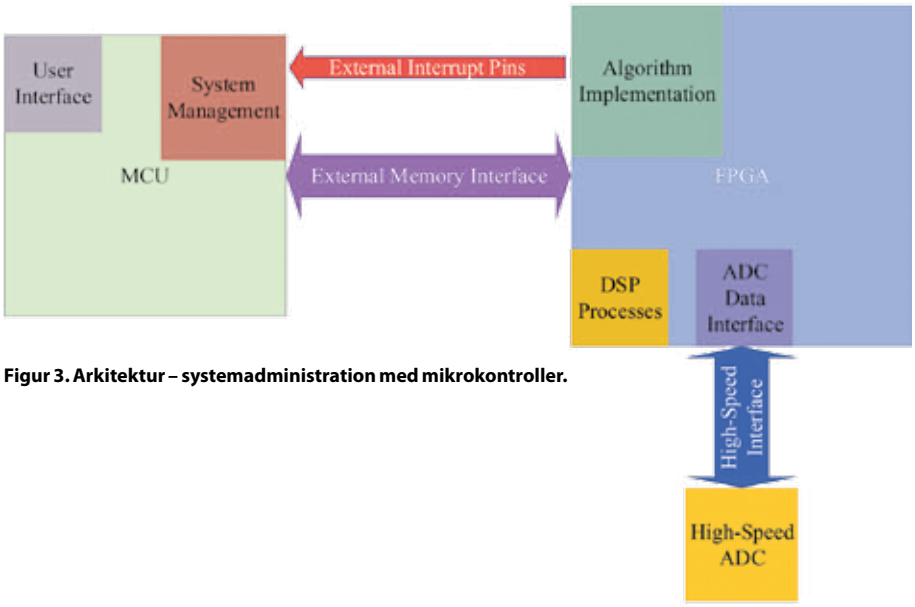
Tidsbegränsningar, processparametrar och andra användarpreferenser måste fortfarande definieras och implementeras, men kärnfunktionaliteten bevaras och översätts till FPGA-kod.

I denna fas är MCU:ns uppgift att ta hand om systemet. Status- och styrregister i FPGA:n övervakas, uppdateras och rapporteras av MCU:n. Den sköter dessutom användargränssnittet (UI), som exempelvis kan köras i en webbserver som nås via en Ethernet- eller wifi-anslutning, eller i en industriell pekskärm.

Den viktigaste lärdomen här är att både MCU och FPGA utnyttjas för de uppgifter som de är bäst lämpade för.

DE VIKTIGASTE RESULTATEN från den andra fasen:

1. FPGA:n hanterar snabb, parallell exekvering av DSP-processer och algoritmer. MCU:n implementerar ett responsivt och smidigt användargränssnitt och hanterar produktens olika processer.
2. Genom att först ha utvecklats och validerats i en MCU har risken för algoritmfel minimerats. Därefter har koden översatts till syntetiserbar HDL. Risken för FPGA-specifika fel kan minimeras exempelvis med hjälp av utvecklingsverktyget Vivado.
3. Det är relativt riskfritt att flytta processer till FPGA. Flera projektintressenter kan därefter direkt se nyttan av FPGA:ns snabbhet och parallellitet. Mätbara prestandaförbättringar observeras och nu kan fokus läggas på att förbereda konstruktionen för tillverkning.



Figur 3. Arkitektur – systemadministration med mikrokontroller.

TREDJE FASEN: Utrullning

Med beräkningsintensiv databehandling i FPGA:n och system och användargränssnitt i MCU:n är produkten redo att tas i bruk. Det här betyder inte att alfa- och beta-releaser kan hoppas över utan poängen vi vill visa är de möjligheter som co-processorarkitekturen ger för utrullning.

Både MCU och FPGA kan uppdateras i fält. Utveckling inom flera områden har gjort att det idag är lika lätt att uppdatera FPGA som att uppdatera mjukvara. Eftersom FPGA:n är adresserbar via MCU:ns minne, kan MCU:n senare fungera för åtkomst till hela systemet: den kan ta emot uppdateringar både för sig själv och för FPGA:n.

Uppdateringar kan schemaläggas, distribueras och anpassas för varje slutanvändare. Slutligen kan loggar över specifika användare och användningsfall upprätthållas och associeras med specifika systemversioner. Med hjälp av dessa data kan prestandan förbättras ytterligare även efter att produkten tagits i skarpt bruk.

NÄR PRODUKTEN ÄR I DRIFT kan uppdateringar och underhåll ske på distans – fördelarna med detta framgår kanske som tydligast när det handlar om rymdbaserade tillämpningar. Det kan handla om något så enkelt som att ändra ett logiskt villkor eller något så komplicerat som att uppdatera ett moduleringsschema för kommunikationen – co-processorarkitekturens programmerbarhet omfattar hela detta spann av funktionalitet. Samtidigt finns flera strålningshärdiga komponenter att välja mellan.

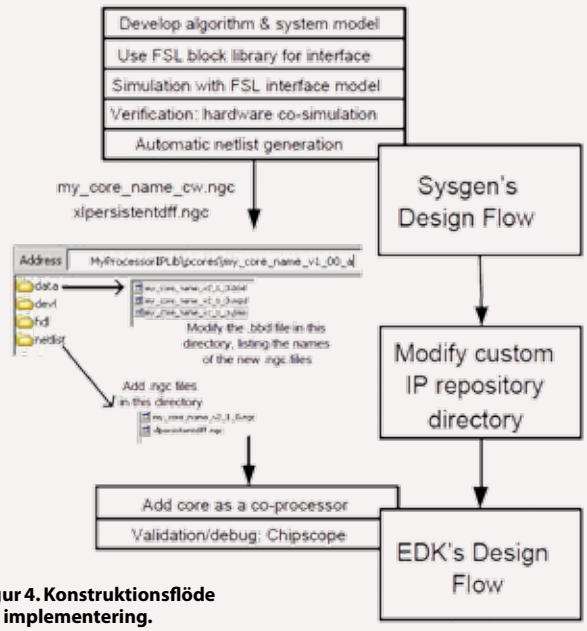
Det viktigaste resultatet från denna fas är den progressiva kostnadsminskningen.

FAKTA

Optimeringsinställningar vid syntetisering i Vivado HLS

När DCT-koden har skapats som ett projekt i Vivado HLS-verktyget är nästa steg att syntetisera konstruktionen för FPGA-implementering. Då märks några av de mest påtagliga fördelarna med att flytta algoritmens exekvering från en MCU till en FPGA tydligare.

I verktyget finns möjligheter till en rad optimeringar:
Standard: C-algoritmen översätts till syntetiserbar HDL utan optimeringar. Detta kan användas som en referens för att bättre förstå de andra optimeringarna.
Inre loop i pipeline: instruerar Vivado HLS att rulla upp inre loopar så att ny data kan börja behandlas medan befintlig data fortfarande befinner sig i pipeline. Nya data behöver alltså inte vänta på att befintliga är färdiga innan de kan börja bearbetas.
Yttre loop i pipeline: den yttre loopens operationer strömmas genom en pipeline. Den inre loopens operationer exekveras jämlöpande. Både latens- och intervalltid halveras genom detta direktiv.
Arraypartitionering: Detta direktiv mappar innehållet i looparna till arrayer och plattar därmed ut all minnesåtkomst till enskilda element i dessa arrayer. Det kräver mer RAM, men körtiden halveras.
Dataflöde: Detta direktiv gör det möjligt för konstruktören att specificera önskat antal klockcykler mellan varje inläsning. Endast loopar och funktioner som processas på toppnivån omfattas av detta direktiv.
Inline: Detta direktiv rullar upp alla loopar, både inre och yttre. Det innebär att looparna tas bort och ersätts av en sekvens kopior av loopkroppen, en per loopvarv. Både rad- och kolumnprocesser kan därmed köras jämlöpande. Antalet nödvändiga klockcykler hålls till ett minimum, även om det förbrukas mer FPGA-resurser.



Figur 4. Konstruktionsflöde för implementering.

Dessutom kan förändringar i komponentlistor och andra optimeringar också komma att göras eftersom det vid sjösättning kan visa sig att produkten fungerar precis lika bra med en billigare MCU eller en enklare FPGA.

Tack vare co-processorn har konstruktörerna inte låst sig till att använda komponenter med överdriven prestanda. Och om en komponent skulle bli otillgänglig går det att byta ut den.

Detta skulle inte vara möjligt om konstruktionen var baserad på ett fullt integrerat chip, en SoC, eller om den försökte hantera all signalbehandling i en högpresterande DSP eller MCU. Co-processorarkitekturen ger en utmärkt mix av kapacitet och flexibilitet som ger konstruktören fler valmöjligheter och friheter, både under utveckling och efter driftsättning.

Snabb prototypframställning

Poängen med att snabbt kunna ta fram prototyper är att det gör det möjligt att täcka in en stor del av produktutvecklingen genom att utföra uppgifter parallellt, snabbt identifiera buggar och konstruktionsproblem och validera kritiska data- och signalvägar.

För att nå ett bra resultat krävs deltagande av expertis inom respektive delområden. Traditionellt innebar det en maskiningenjör, en mjukvaruingenjör och en HDL-utvecklare. Numera finns det dock gott om tvärvetenskapliga yrkesverksamma som kan fylla olika roller, även om projektkostnaden för att samordna dessa insatser fortfarande är betydande.

En forskningsartikel med titeln ”An FPGA based rapid prototyping platform for wavelet coprocessors” lyfter fram hur co-processorarkitekturen gjorde det möjligt för en ingenjör att fylla alla dessa roller på ett effektivt sätt.

Studien började med att forskarna konstruerade och simulerade den önskade DSP-funktionaliteten i det Matlab-baserade utvecklingsverktyg Simulink. Det gjordes av två huvudanledningar:

- det verifierade den önskade prestandan genom simulering.
- det fungerade som en grundnivå, som framtida konstruktioner kunde jämföras med och referera till.

DÄREFTER IDENTIFIERADES kritiska funktioner som delades upp i olika kärnor – mjukvarukomponenter och delar som kunde syntetiseras i en FPGA. Det viktigaste här var att definiera gränssnittet mellan de olika kärnorna och komponenterna och att jämföra dataprestanda med simuleringen. Denna konstruktionsprocess stämde väl överens med Xilinx konstruktionsflöde för inbyggda system, som sammanfattas i figur 4.

Genom att dela upp systemet i syntetiserbara kärnor kan DSP-konstruktören fokusera på de mest kritiska delarna av signalbehandlingen. Hen behöver inte vara expert på hårdvara eller HDL för att modifiera, routa eller implementera olika mjuka processorkärnor eller komponenter i FPGA:n. Så länge som konstruktören är medveten om gränssnittet och dataformaten har hen full kontroll över signalvägarna och kan förbättra systemets prestanda. ■

FALLSTUDIE

Diskret cosinustransformation

Diskret cosinustransformation (DCT) används frekvent inom digital signalbehandling för mönsterigenkänning och -filtrering. Författaren och hans kollegor valde därför i en labbövning ut DCT som exempel på en beräkningsintensiv algoritm. Den röda tråden är hur algoritmen utvecklas från en C- till en HDL-implementering.

Följande verktyg och komponenter användes:
- Vivado HLS v2019
- Den specifika komponent vi använde för utvärdering och simulering har produktnummer xczu7ev-ffvc1156-2-e

DCT-algoritmen – till att börja med i C – arbetar med två arrayer av 16-bitars heltal: array a är indata och array b är utdata. Databredden (DW) definieras därför som 16, medan antalet element (N) i arrayerna är 1024/DW, det vill säga 64. Slutligen är storleken på DCT-arrayen (DCT_SIZE) satt till 8, vilket betyder att vi använder en 8x8-matris.

Den C-baserade implementeringen gör att det går snabbt att utveckla och validera algoritmens funktion. Vi väljer att låta funktion vara viktigare än prestanda eftersom eftersom den senare implementeringen i en FPGA enkelt kommer att kunna trimma prestanda genom att utnyttja hårdvaruacceleration, loop unrolling och annan teknik.

Figur 5. Xilinx Vivado HLS-konstruktionsflöde.